man pages section 9: DDI and DKI Properties and Data Structures

Part No: 816–5181–17
August 2011

111219@25097

Contents

Preface

Both novice users and those familar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question "What does it do?" The man pages in general comprise a reference manual. They are not intended to be a tutorial.

Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man(1)` for more information about man pages in general.

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.
SYNOPSIS	This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]	Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
. . .	Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "`filename...`".
|	Separator. Only one of the arguments separated by this character can be specified at a time.
{ }	Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL	This section occurs only in subsection 3R to indicate the protocol description file.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.
IOCTL	This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the `ioctl(2)` system call is called `ioctl` and generates its own

heading. `ioctl` calls for a specific device are listed alphabetically (on the man page for that specific device). `ioctl` calls are used for a particular class of devices all of which have an `io` ending, such as `mtio`(7I).

OPTIONS	This secton lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output – standard output, standard error, or output files – generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.
USAGE	This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar

EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.
FILES	This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
ATTRIBUTES	This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes(5)` for more information.
SEE ALSO	This section lists references to other man pages, in-house documentation, and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.
BUGS	This section describes known bugs and, wherever possible, suggests workarounds.

REFERENCE

Introduction

Name Intro – introduction to kernel data structures and properties

Description Section 9P describes kernel properties used by device drivers. Section 9S describes the data structures used by drivers to share information between the driver and the kernel. See Intro(9E) for an overview of device driver interfaces.

In Section 9S, reference pages contain the following headings:

- NAME summarizes the purpose of the structure or property.
- SYNOPSIS lists the include file that defines the structure or property.
- INTERFACE LEVEL describes any architecture dependencies.
- DESCRIPTION provides general information about the structure or property.
- STRUCTURE MEMBERS lists all accessible structure members (for Section 9S).
- SEE ALSO gives sources for further information.

Of the preceding headings, Section 9P reference pages contain the NAME, DESCRIPTION, and SEE ALSO fields.

Every driver MUST include <sys/ddi.h> and <sys/sunddi.h>, in that order, and as final entries.

The following table summarizes the STREAMS structures described in Section 9S.

Structure	Type
copyreq	DDI/DKI
copyresp	DDI/DKI
datab	DDI/DKI
fmodsw	Solaris DDI
free_rtn	DDI/DKI
iocblk	DDI/DKI
linkblk	DDI/DKI
module_info	DDI/DKI
msgb	DDI/DKI
qband	DDI/DKI
qinit	DDI/DKI
queclass	Solaris DDI
queue	DDI/DKI
streamtab	DDI/DKI

Structure	Type
stroptions	DDI/DKI

The following table summarizes structures that are not specific to STREAMS I/O.

Structure	Type
aio_req	Solaris DDI
buf	DDI/DKI
cb_ops	Solaris DDI
ddi_device_acc_attr	Solaris DDI
ddi_dma_attr	Solaris DDI
ddi_dma_cookie	Solaris DDI
ddi_dma_lim_sparc	Solaris SPARC DDI
ddi_dma_lim_x86	Solaris x86 DDI
ddi_dma_req	Solaris DDI
ddi_dmae_req	Solaris x86 DDI
ddi_idevice_cookie	Solaris DDI
ddi_mapdev_ctl	Solaris DDI
devmap_callback_ctl	Solaris DDI
dev_ops	Solaris DDI
iovec	DDI/DKI
kstat	Solaris DDI
kstat_intr	Solaris DDI
kstat_io	Solaris DDI
kstat_named	Solaris DDI
map	DDI/DKI
modldrv	Solaris DDI
modlinkage	Solaris DDI
modlstrmod	Solaris DDI
scsi_address	Solaris DDI

Structure	Type
scsi_arq_status	Solaris DDI
scsi_device	Solaris DDI
scsi_extended_sense	Solaris DDI
scsi_hba_tran	Solaris DDI
scsi_inquiry	Solaris DDI
scsi_pkt	Solaris DDI
scsi_status	Solaris DDI
uio	DDI/DKI

See Also Intro(9E)

Notes Do not declare arrays of structures as the size of the structures can change between releases. Rely only on the structure members listed in this chapter and not on unlisted members or the position of a member in a structure.

REFERENCE

Data Structures for Drivers

Name aio_req – asynchronous I/O request structure

Synopsis

```
#include <sys/uio.h>
#include <sys/aio_req.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description An `aio_req` structure describes an asynchronous I/O request.

Structure Members

```
struct uio*aio_uio;   /* uio structure describing the I/O request */
```

The `aio_uio` member is a pointer to a `uio(9S)` structure, describing the I/O transfer request.

See Also `aread(9E)`, `awrite(9E)`, `aphysio(9F)`, `uio(9S)`

Name buf – block I/O data transfer structure

Synopsis

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Architecture independent level 1 (DDI/DKI)

Description The `buf` structure is the basic data structure for block I/O transfers. Each block I/O transfer has an associated buffer header. The header contains all the buffer control and status information. For drivers, the buffer header pointer is the sole argument to a block driver `strategy`(9E) routine. Do not depend on the size of the `buf` structure when writing a driver.

A buffer header can be linked in multiple lists simultaneously. Because of this, most of the members in the buffer header cannot be changed by the driver, even when the buffer header is in one of the driver's work lists.

Buffer headers are also used by the system for unbuffered or physical I/O for block drivers. In this case, the buffer describes a portion of user data space that is locked into memory.

Block drivers often chain block requests so that overall throughput for the device is maximized. The `av_forw` and the `av_back` members of the `buf` structure can serve as link pointers for chaining block requests.

Structure Members

```
int             b_flags;            /* Buffer status */
struct buf      *av_forw;           /* Driver work list link */
struct buf      *av_back;           /* Driver work list link */
size_t          b_bcount;           /* # of bytes to transfer */
union {
     caddr_t    b_addr;             /* Buffer’s virtual address */
} b_un;
daddr_t         b_blkno;            /* Block number on device */
diskaddr_t      b_lblkno;           /* Expanded block number on dev. */
size_t          b_resid;            /* # of bytes not xferred */
size_t          b_bufsize;          /* size of alloc. buffer */
int             (*b_iodone)(struct buf *); /* function called */
                                           /* by biodone */
int             b_error;            /* expanded error field */
void            *b_private;         /* "opaque" driver private area */
dev_t           b_edev;             /* expanded dev field */
```

The members of the buffer header available to test or set by a driver are as follows:

`b_flags` stores the buffer status and indicates to the driver whether to read or write to the device. The driver must never clear the `b_flags` member. If this is done, unpredictable results can occur including loss of disk sanity and the possible failure of other kernel processes.

All `b_flags` bit values not otherwise specified above are reserved by the kernel and may not be used.

Valid flags are as follows:

`B_BUSY`	Indicates the buffer is in use. The driver must not change this flag unless it allocated the buffer with `getrbuf(9F)` and no I/O operation is in progress.
`B_DONE`	Indicates the data transfer has completed. This flag is read-only.
`B_ERROR`	Indicates an I/O transfer error. It is set in conjunction with the `b_error` field. `bioerror(9F)` should be used in preference to setting the `B_ERROR` bit.
`B_PAGEIO`	Indicates the buffer is being used in a paged I/O request. See the description of the `b_un.b_addr` field for more information. This flag is read-only.
`B_PHYS`	indicates the buffer header is being used for physical (direct) I/O to a user data area. See the description of the `b_un.b_addr` field for more information. This flag is read-only.
`B_READ`	Indicates that data is to be read from the peripheral device into main memory.
`B_WRITE`	Indicates that the data is to be transferred from main memory to the peripheral device. `B_WRITE` is a pseudo flag and cannot be directly tested; it is only detected as the NOT form of `B_READ`.

`av_forw` and `av_back` can be used by the driver to link the buffer into driver work lists.

`b_bcount` specifies the number of bytes to be transferred in both a paged and a non-paged I/O request.

`b_un.b_addr` is the virtual address of the I/O request, unless `B_PAGEIO` is set. The address is a kernel virtual address, unless `B_PHYS` is set, in which case it is a user virtual address. If `B_PAGEIO` is set, `b_un.b_addr` contains kernel private data. Note that either one of `B_PHYS` and `B_PAGEIO`, or neither, can be set, but not both.

`b_blkno` identifies which logical block on the device (the device is defined by the device number) is to be accessed. The driver might have to convert this logical block number to a physical location such as a cylinder, track, and sector of a disk. This is a 32-bit value. The driver should use `b_blkno` or `b_lblkno`, but not both.

`b_lblkno` identifies which logical block on the device (the device is defined by the device number) is to be accessed. The driver might have to convert this logical block number to a physical location such as a cylinder, track, and sector of a disk. This is a 64-bit value. The driver should use `b_lblkno` or `b_blkno`, but not both.

`b_resid` should be set to the number of bytes not transferred because of an error.

`b_bufsize` contains the size of the allocated buffer.

`b_iodone` identifies a specific `biodone` routine to be called by the driver when the I/O is complete.

b_error can hold an error code that should be passed as a return code from the driver. b_error is set in conjunction with the B_ERROR bit set in the b_flags member. bioerror(9F) should be used in preference to setting the b_error field.

b_private is for the private use of the device driver.

b_edev contains the major and minor device numbers of the device accessed.

See Also strategy(9E), aphysio(9F), bioclone(9F), biodone(9F), bioerror(9F), bioinit(9F), clrbuf(9F), getrbuf(9F), physio(9F), iovec(9S), uio(9S)

Writing Device Drivers

Warnings Buffers are a shared resource within the kernel. Drivers should read or write only the members listed in this section. Drivers that attempt to use undocumented members of the buf structure risk corrupting data in the kernel or on the device.

Name cb_ops – character/block entry points structure

Synopsis

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `cb_ops` structure contains all entry points for drivers that support both character and block entry points. All leaf device drivers that support direct user process access to a device should declare a `cb_ops` structure.

All drivers that safely allow multiple threads of execution in the driver at the same time must set the `D_MP` flag in the `cb_flag` field. See `open`(9E).

If the driver properly handles 64-bit offsets, it should also set the `D_64BIT` flag in the `cb_flag` field. This specifies that the driver will use the `uio_loffset` field of the `uio`(9S) structure.

If the driver returns `EINTR` from `open`(9E), it should also set the `D_OPEN_RETURNS_EINTR` flag in the `cb_flag` field. This lets the framework know that it is safe for the driver to return `EINTR` when waiting, to provide exclusion for a last-reference `close`(9E) call to complete before calling `open`(9E).

The `mt-streams`(9F) function describes other flags that can be set in the `cb_flag` field.

The `cb_rev` is the `cb_ops` structure revision number. This field must be set to `CB_REV`.

Non-STREAMS drivers should set `cb_str` to `NULL`.

The following DDI/DKI or DKI-only or DDI-only functions are provided in the character/block driver operations structure.

block/char	Function	Description
b/c	XXopen	DDI/DKI
b/c	XXclose	DDI/DKI
b	XXstrategy	DDI/DKI
b	XXprint	DDI/DKI
b	XXdump	DDI(Sun)
c	XXread	DDI/DKI
c	XXwrite	DDI/DKI
c	XXioctl	DDI/DKI

block/char	Function	Description
c	XXdevmap	DDI(Sun)
c	XXmmap	DKI
c	XXsegmap	DKI
c	XXchpoll	DDI/DKI
c	XXprop_op	DDI(Sun)
c	XXaread	DDI(Sun)
c	XXawrite	DDI(Sun)

Structure Members

```
int  (*cb_open)(dev_t *devp, int flag, int otyp, cred_t *credp);
int  (*cb_close)(dev_t dev, int flag, int otyp, cred_t *credp);
int  (*cb_strategy)(struct buf *bp);
int  (*cb_print)(dev_t dev, char *str);
int  (*cb_dump)(dev_t dev, caddr_t addr, daddr_t blkno, int nblk);
int  (*cb_read)(dev_t dev, struct uio *uiop, cred_t *credp);
int  (*cb_write)(dev_t dev, struct uio *uiop, cred_t *credp);
int  (*cb_ioctl)(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *credp, int *rvalp);
int  (*cb_devmap)(dev_t dev, devmap_cookie_t dhp, offset_t off,
        size_t len, size_t *maplen, uint_t model);
int  (*cb_mmap)(dev_t dev, off_t off, int prot);
int  (*cb_segmap)(dev_t dev, off_t off, struct as *asp,
        caddr_t *addrp, off_t len, unsigned int prot,
        unsigned int maxprot, unsigned int flags, cred_t *credp);
int  (*cb_chpoll)(dev_t dev, short events, int anyyet,
        short *reventsp, struct pollhead **phpp);
int  (*cb_prop_op)(dev_t dev, dev_info_t *dip,
        ddi_prop_op_t prop_op, int mod_flags,
        char *name, caddr_t valuep, int *length);
struct streamtab *cb_str;   /* streams information */
int  cb_flag;
int  cb_rev;
int  (*cb_aread)(dev_t dev, struct aio_req *aio, cred_t *credp);
int  (*cb_awrite)(dev_t dev, struct aio_req *aio, cred_t *credp);
```

See Also

aread(9E), awrite(9E), chpoll(9E), close(9E), dump(9E), ioctl(9E), mmap(9E), open(9E), print(9E), prop_op(9E), read(9E), segmap(9E), strategy(9E), write(9E), nochpoll(9F), nodev(9F), nulldev(9F), dev_ops(9S), qinit(9S)

Writing Device Drivers

STREAMS Programming Guide

Name copyreq – STREAMS data structure for the M_COPYIN and the M_COPYOUT message types

Synopsis

```
#include <sys/stream.h>
```

Interface Level Architecture independent level 1 (DDI/DKI)

Description The data structure for the M_COPYIN and the M_COPYOUT message types.

Structure Members

```
int      cq_cmd;           /* ioctl command (from ioc_cmd) */
cred_t   *cq_cr;           /* full credentials */
uint_t   cq_id;            /* ioctl id (from ioc_id) */
uint_t   cq_flag;           /* must be zero */
mblk_t   *cq_private;      /* private state information */
caddr_t  cq_addr;          /* address to copy data to/from */
size_t   cq_size;          /* number of bytes to copy */
```

See Also *STREAMS Programming Guide*

Name copyresp – STREAMS data structure for the M_IOCDATA message type

Synopsis `#include <sys/stream.h>`

Interface Level Architecture independent level 1 (DDI/DKI)

Description The data structure copyresp is used with the `M_IOCDATA` message type.

Structure Members

```
int      cp_cmd;       /* ioctl command (from ioc_cmd) */
cred_t   *cp_cr;       /* full credentials */
uint_t   cp_id;        /* ioctl id (from ioc_id) */
uint_t   cp_flag;      /* ioctl flags */
mblk_t   *cp_private;  /* private state information */
caddr_t  cp_rval;      /* status of request: 0 -> success;
                       /* non-zero -> failure */
```

See Also *STREAMS Programming Guide*

Name datab, dblk – STREAMS message data structure

Synopsis `#include <sys/stream.h>`

Interface Level Architecture independent level 1 (DDI/DKI).

Description The `datab` structure describes the data of a STREAMS message. The actual data contained in a STREAMS message is stored in a data buffer pointed to by this structure. A `msgb` (message block) structure includes a field that points to a `datab` structure.

Because a data block can have more than one message block pointing to it at one time, the `db_ref` member keeps track of a data block's references, preventing it from being deallocated until all message blocks are finished with it.

Structure Members

```
unsigned char     *db_base;     /* first byte of buffer */
unsigned char     *db_lim;      /* last byte (+1) of buffer */
unsigned char      db_ref;      /* # of message pointers to this data */
unsigned char      db_type;     /* message type */
```

A `datab` structure is defined as type `dblk_t`.

See Also `free_rtn`(9S), `msgb`(9S)

Writing Device Drivers

STREAMS Programming Guide

Name ddi_device_acc_attr – data access attributes structure

Synopsis

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `ddi_device_acc_attr` structure describes the data access characteristics and requirements of the device.

Structure Members

```
ushort_t     devacc_attr_version;
uchar_t      devacc_attr_endian_flags;
uchar_t      devacc_attr_dataorder;
uchar_t      devacc_attr_access;
```

The `devacc_attr_version` member identifies the version number of this structure. The current version number is `DDI_DEVICE_ATTR_V0`.

The `devacc_attr_endian_flags` member describes the endian characteristics of the device. Specify one of the following values:

`DDI_NEVERSWAP_ACC` Data access with no byte swapping

`DDI_STRUCTURE_BE_ACC` Structural data access in big-endian format

`DDI_STRUCTURE_LE_ACC` Structural data access in little endian format

`DDI_STRUCTURE_BE_ACC` and `DDI_STRUCTURE_LE_ACC` describe the endian characteristics of the device as big-endian or little-endian, respectively. Although most of the devices have the same endian characteristics as their buses, examples of devices that have opposite endian characteristics of the buses do exist. When `DDI_STRUCTURE_BE_ACC` or `DDI_STRUCTURE_LE_ACC` is set, byte swapping is automatically performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation can take advantage of hardware platform byte swapping capabilities.

When you specify `DDI_NEVERSWAP_ACC`, byte swapping is not invoked in the data access functions.

The `devacc_attr_dataorder` member describes the order in which the CPU references data. Specify one of the following values.

`DDI_STRICTORDER_ACC` Data references must be issued by a CPU in program order. Strict ordering is the default behavior.

`DDI_UNORDERED_OK_ACC` The CPU can reorder the data references. This includes all kinds of reordering. For example, a load followed by a store might be replaced by a store followed by a load.

`DDI_MERGING_OK_ACC` The CPU can merge individual stores to consecutive locations. For example, the CPU can turn two consecutive byte stores into one half-word store. It can also batch individual loads. For example, the CPU might turn two consecutive byte loads into one half-word load. DDI_MERGING_OK_ACC also implies reordering.

DDI_LOADCACHING_OK_ACC
: The CPU can cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. DDI_LOADCACHING_OK_ACC also implies merging and reordering.

DDI_STORECACHING_OK_ACC
: The CPU can keep the data in the cache and push it to the device, perhaps with other data, at a later time. The default behavior is to push the data right away. DDI_STORECACHING_OK_ACC also implies load caching, merging, and reordering.

These values are advisory, not mandatory. For example, data can be ordered without being merged, or cached, even though a driver requests unordered, merged, and cached together.

The values defined for devacc_attr_access are:

DDI_DEFAULT_ACC
: If an I/O fault occurs, the system will take the default action, which might be to panic.

DDI_FLAGERR_ACC
: Using this value indicates that the driver is hardened: able to cope with the incorrect results of I/O operations that might result from an I/O fault. The value also indicates that the driver will use ddi_fm_acc_err_get(9F) to check access handles for faults on a regular basis.

: If possible, the system should not panic on such an I/O fault, but should instead mark the I/O handle through which the access was made as having faulted.

: This value is advisory: it tells the system that the driver can continue in the face of I/O faults. The value does not guarantee that the system will not panic, as that depends on the nature of the fault and the capabilities of the system. It is quite legitimate for an implementation to ignore this flag and panic anyway.

DDI_CAUTIOUS_ACC
: This value indicates that an I/O fault is anticipated and should be handled as gracefully as possible. For example, the framework should not print a console message.

: This value should be used when it is not certain that a device is physically present: for example, when probing. As such, it provides an alternative within the DDI access framework to the existing peek/poke functions, which don't use access handles and cannot be integrated easily into a more general I/O fault handling framework.

In order to guarantee safe recovery from an I/O fault, it might be necessary to acquire exclusive access to the parent bus, for example, or to synchronize across processors on an MP machine. “Cautious” access can be quite expensive and is only recommended for initial probing and possibly for additional fault-recovery code.

Examples

The following examples illustrate the use of device register address mapping setup functions and different data access functions.

EXAMPLE 1 Using `ddi_device_acc_attr()` in >`ddi_regs_map_setup`(9F)

This example demonstrates the use of the `ddi_device_acc_attr()` structure in `ddi_regs_map_setup`(9F). It also shows the use of `ddi_getw`(9F) and `ddi_putw`(9F) functions in accessing the register contents.

```
dev_info_t *dip;
uint_t     rnumber;
ushort_t  *dev_addr;
offset_t   offset;
offset_t   len;
ushort_t   dev_command;
ddi_device_acc_attr_t dev_attr;
ddi_acc_handle_t handle;


. . .


/*
 * setup the device attribute structure for little endian,
 * strict ordering and 16-bit word access.
 */
dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

/*
 * set up the device registers address mapping
 */
ddi_regs_map_setup(dip, rnumber, (caddr_t *)&dev_addr, offset, len,
        &dev_attr, &handle);

/* read a 16-bit word command register from the device        */
dev_command = ddi_getw(handle, dev_addr);

dev_command |= DEV_INTR_ENABLE;
/* store a new value back to the device command register    */
```

EXAMPLE 1 Using `ddi_device_acc_attr()` in >`ddi_regs_map_setup`(9F) *(Continued)*

```
ddi_putw(handle, dev_addr, dev_command);
```

EXAMPLE 2 Accessing a Device with Different Apertures

The following example illustrates the steps used to access a device with different apertures. Several apertures are assumed to be grouped under one single "reg" entry. For example, the sample device has four different apertures, each 32 Kbyte in size. The apertures represent YUV little-endian, YUV big-endian, RGB little-endian, and RGB big-endian. This sample device uses entry 1 of the "reg" property list for this purpose. The size of the address space is 128 Kbyte with each 32 Kbyte range as a separate aperture. In the register mapping setup function, the sample driver uses the *offset* and *len* parameters to specify one of the apertures.

```
ulong_t    *dev_addr;
ddi_device_acc_attr_t dev_attr;
ddi_acc_handle_t handle;
uchar_t buf[256];

. . .

/*
 * setup the device attribute structure for never swap,
 * unordered and 32-bit word access.
 */
dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_attr.devacc_attr_endian_flags = DDI_NEVERSWAP_ACC;
dev_attr.devacc_attr_dataorder = DDI_UNORDERED_OK_ACC;

/*
 * map in the RGB big-endian aperture
 * while running in a big endian machine
 *  - offset 96K and len 32K
 */
ddi_regs_map_setup(dip, 1, (caddr_t *)&dev_addr, 96*1024, 32*1024,
        &dev_attr, &handle);

/*
 * Write to the screen buffer
 *  first 1K bytes words, each size 4 bytes
 */
ddi_rep_putl(handle, buf, dev_addr, 256, DDI_DEV_AUTOINCR);
```

EXAMPLE 3 Functions That Call Out the Data Word Size

The following example illustrates the use of the functions that explicitly call out the data word size to override the data size in the device attribute structure.

EXAMPLE 3 Functions That Call Out the Data Word Size *(Continued)*

```
struct device_blk {
    ushort_t    d_command;    /* command register */
    ushort_t    d_status;    /* status register */
    ulong        d_data;        /* data register */
} *dev_blkp;
dev_info_t *dip;
caddr_t    dev_addr;
ddi_device_acc_attr_t dev_attr;
ddi_acc_handle_t handle;
uchar_t buf[256];

. . .

/*
 * setup the device attribute structure for never swap,
 * strict ordering and 32-bit word access.
 */
dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_attr.devacc_attr_endian_flags = DDI_NEVERSWAP_ACC;
dev_attr.devacc_attr_dataorder= DDI_STRICTORDER_ACC;

ddi_regs_map_setup(dip, 1, (caddr_t *)&dev_blkp, 0, 0,
        &dev_attr, &handle);

/* write command to the 16-bit command register */
ddi_putw(handle, &dev_blkp->d_command, START_XFER);

/* Read the 16-bit status register */
status = ddi_getw(handle, &dev_blkp->d_status);

if (status & DATA_READY)
        /* Read 1K bytes off the 32-bit data register */
        ddi_rep_getl(handle, buf, &dev_blkp->d_data,
                256, DDI_DEV_NO_AUTOINCR);
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

attributes(5), ddi_fm_acc_err_get(9F), ddi_getw(9F), ddi_putw(9F), ddi_regs_map_setup(9F)

Writing Device Drivers

Name ddi_dma_attr – DMA attributes structure

Synopsis

```
#include <sys/ddidmareq.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description A `ddi_dma_attr_t` structure describes device– and DMA engine-specific attributes necessary to allocate DMA resources for a device. The driver might have to extend the attributes with bus-specific information, depending on the bus to which the device is connected.

Structure Members

```
uint_t      dma_attr_version;     /* version number */
uint64_t    dma_attr_addr_lo;     /* low DMA address range */
uint64_t    dma_attr_addr_hi;     /* high DMA address range */
uint64_t    dma_attr_count_max;   /* DMA counter register */
uint64_t    dma_attr_align;       /* DMA address alignment */
uint_t      dma_attr_burstsizes;  /* DMA burstsizes */
uint32_t    dma_attr_minxfer;     /* min effective DMA size */
uint64_t    dma_attr_maxxfer;     /* max DMA xfer size */
uint64_t    dma_attr_seg;         /* segment boundary */
int         dma_attr_sgllen;      /* s/g list length */
uint32_t    dma_attr_granular;    /* granularity of device */
uint_t      dma_attr_flags;       /* DMA transfer flags */
```

The `dma_attr_version` stores the version number of this DMA attribute structure. It should be set to `DMA_ATTR_V0`.

The `dma_attr_addr_lo` and `dma_attr_addr_hi` fields specify the address range the device's DMA engine can access. The `dma_attr_addr_lo` field describes the inclusive lower 64–bit boundary. The `dma_attr_addr_hi` describes the inclusive upper 64–bit boundary. The system ensures that allocated DMA resources are within the range specified. See `ddi_dma_cookie`(9S).

The `dma_attr_count_max` describes an inclusive upper bound for the device's DMA counter register. For example, 0xFFFFFF would describe a DMA engine with a 24–bit counter register. DMA resource allocation functions have to break up a DMA object into multiple DMA cookies if the size of the object exceeds the size of the DMA counter register.

The `dma_attr_align` specifies alignment requirements for allocated DMA resources. This field can be used to force more restrictive alignment than imposed by `dma_attr_burstsizes` or `dma_attr_minxfer`, such as alignment at a page boundary. Most drivers set this field to 1, indicating byte alignment.

The `dma_attr_align` only specifies alignment requirements for allocated DMA resources. The buffer passed to `ddi_dma_addr_bind_handle`(9F) or `ddi_dma_buf_bind_handle`(9F) must have an equally restrictive alignment (see `ddi_dma_mem_alloc`(9F)).

The `dma_attr_burstsizes` field describes the possible burst sizes the DMA engine of a device can accept. The format of the data sizes is binary, encoded in terms of powers of two. When

DMA resources are allocated, the system can modify the burstsizes value to reflect the system limits. The driver must use the allowable burstsizes to program the DMA engine. See ddi_dma_burstsizes(9F).

The dma_attr_minxfer field describes the minimum effective DMA access size in units of bytes. DMA resources can be modified, depending on the presence and use of I/O caches and write buffers between the DMA engine and the memory object. This field is used to determine alignment and padding requirements for ddi_dma_mem_alloc(9F).

The dma_attr_maxxfer field describes the maximum effective DMA access size in units of bytes.

The dma_attr_seg field specifies segment boundary restrictions for allocated DMA resources. The system allocates DMA resources for the device so that the object does not span the segment boundary specified by dma_attr_seg. For example, a value of 0xFFFF means DMA resources must not cross a 64–Kbyte boundary. DMA resource allocation functions might have to break up a DMA object into multiple DMA cookies to enforce segment boundary restrictions. In this case, the transfer must be performed using scatter-gather I/O or multiple DMA windows.

The dma_attr_sgllen field describes the length of the DMA scatter/gather list of a device. Possible values are as follows:

< 0 Device DMA engine is not constrained by the size, for example, withDMA chaining.

= 0 Reserved.

= 1 Device DMA engine does not support scatter/gather such as third party DMA.

> 1 Device DMA engine uses scatter/gather. The dma_attr_sgllen value is the maximum number of entries in the list.

The dma_attr_granular field describes the granularity of the device transfer size in units of bytes. When the system allocates DMA resources, the size of a single segment is a multiple of the device granularity. If dma_attr_sgllen is larger than 1 within a window, the sum of the sizes for a subgroup of segments is a multiple of the device granularity.

All driver requests for DMA resources must be a multiple of the granularity of the device transfer size.

The dma_attr_flags field can be set to a combination of:

DDI_DMA_FORCE_PHYSICAL Some platforms, such as SPARC systems, support what is called Direct Virtual Memory Access (DVMA). On these platforms, the device is provided with a virtual address by the system in order to perform the transfer. In this case, the underlying platform provides an *IOMMU*, which translates accesses to these virtual addresses into the proper physical addresses. Some of these platforms also support DMA.

DDI_DMA_FORCE_PHYSICAL indicates that the system should return physical rather than virtual I/O addresses if the system supports both. If the system does not support physical DMA, the return value from ddi_dma_alloc_handle(9F) is DDI_DMA_BADATTR. In this case, the driver has to clear DDI_DMA_FORCE_PHYSICAL and retry the operation.

DDI_DMA_FLAGERR
: Using this value indicates that the driver is hardened: able to cope with the incorrect results of DMA operations that might result from an I/O fault. The value also indicates that the driver will use ddi_fm_dma_err_get(9F) to check DMA handles for faults on a regular basis.

  If a DMA error is detected during a DMA access to an area mapped by such a handle, the system should not panic if possible, but should instead mark the DMA handle as having faulted.

  This value is advisory: it tells the system that the driver can continue in the face of I/O faults. It does not guarantee that the system will not panic, as that depends on the nature of the fault and the capabilities of the system. It is quite legitimate for an implementation to ignore this flag and panic anyway.

DDI_DMA_RELAXED_ORDERING
: This optional flag can be set if the DMA transactions associated with this handle are not required to observe strong DMA write ordering among themselves, nor with DMA write transactions of other handles.

  The flag allows the host bridge to transfer data to and from memory more efficiently and might result in better DMA performance on some platforms.

  Drivers for devices with hardware support, such as marking the bus transactions relaxed ordered, should not use this flag. Such drivers should use the hardware capability instead.

Examples

EXAMPLE 1 Initializing the ddi_dma_attr_t Structure

Assume a device has the following DMA characteristics:

- Full 32-bit range addressable
- 24-bit DMA counter register
- Byte alignment

EXAMPLE 1 Initializing the `ddi_dma_attr_t` Structure *(Continued)*

- 4– and 8-byte burst sizes support
- Minimum effective transfer size of 1 bytes
- 64 Mbyte maximum transfer size limit
- Maximum segment size of 32 Kbyte
- 17 scatter/gather list elements
- 512–byte device transfer size granularity

The corresponding `ddi_dma_attr_t` structure is initialized as follows:

```
static ddi_dma_attr_t dma_attrs = {
         DMA_ATTR_V0                 /* version number */
         (uint64_t)0x0,              /* low address */
         (uint64_t)0xffffffff,       /* high address */
         (uint64_t)0xffffff,         /* DMA counter max */
         (uint64_t)0x1               /* alignment */
         0x0c,                       /* burst sizes */
         0x1,                        /* minimum transfer size */
         (uint64_t)0x3ffffff,        /* maximum transfer size */
         (uint64_t)0x7fff,           /* maximum segment size */
         17,                         /* scatter/gather list lgth */
         512                         /* granularity */
         0                           /* DMA flags */
};
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also

`attributes(5)`, `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`, `ddi_dma_buf_bind_handle(9F)`, `ddi_dma_burstsizes(9F)`, `ddi_dma_mem_alloc(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_fm_dma_err_get(9F)`, `ddi_dma_cookie(9S)`

Writing Device Drivers

Name ddi_dma_cookie – DMA address cookie

Synopsis

```
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The ddi_dma_cookie_t structure contains DMA address information required to program a DMA engine. The structure is filled in by a call to ddi_dma_getwin(9F), ddi_dma_addr_bind_handle(9F), or ddi_dma_buf_bind_handle(9F), to get device-specific DMA transfer information for a DMA request or a DMA window.

Structure Members

```
typedef struct {
    union {
            uint64_t        _dmac_ll;    /* 64 bit DMA add. */
            uint32_t        _dmac_la[2]; /* 2 x 32 bit add. */
    } _dmu;
    size_t          dmac_size;   /* DMA cookie size */
    uint_t          dmac_type;   /* bus spec. type bits */
} ddi_dma_cookie_t;
```

You can access the DMA address through the #defines: dmac_address for 32-bit addresses and dmac_laddress for 64-bit addresses. These macros are defined as follows:

```
#define dmac_laddress   _dmu._dmac_ll
#ifdef _LONG_LONG_HTOL
#define dmac_notused    _dmu._dmac_la[0]
#define dmac_address    _dmu._dmac_la[1]
#else
#define dmac_address    _dmu._dmac_la[0]
#define dmac_notused    _dmu._dmac_la[1]
#endif
```

dmac_laddress specifies a 64-bit I/O address appropriate for programming the device's DMA engine. If a device has a 64-bit DMA address register a driver should use this field to program the DMA engine. dmac_address specifies a 32-bit I/O address. It should be used for devices that have a 32-bit DMA address register. The I/O address range that the device can address and other DMA attributes have to be specified in a ddi_dma_attr(9S) structure.

dmac_size describes the length of the transfer in bytes.

dmac_type contains bus-specific type bits, if appropriate. For example, a device on a PCI bus has PCI address modifier bits placed here.

See Also pci(4), sbus(4), sysbus(4), ddi_dma_addr_bind_handle(9F), ddi_dma_buf_bind_handle(9F), ddi_dma_getwin(9F), ddi_dma_nextcookie(9F), ddi_dma_attr(9S)

Writing Device Drivers

Name ddi_dmae_req – DMA engine request structure

Synopsis `#include <sys/dma_engine.h>`

Interface Level Solaris x86 DDI specific (Solaris x86 DDI).

Description A device driver uses the `ddi_dmae_req` structure 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 might 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_tder_command;          /* Read / Write *
/uchar_tder_bufprocess;      /* Standard / Chain */
uchar_tder_path;             /* 8 / 16 / 32 */
uchar_tder_cycles;           /* Compat / Type A / Type B / Burst */
uchar_tder_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 can 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. This action causes 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. This routine 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 can specify the DMAE_BUF_CHAIN flag only if the particular bus architecture supports the use of multiple DMA cookies in a single I/O transfer. A bus DMA engine can support this feature either with a fixed-length scatter/gather list, or by an interrupt chaining feature. A driver must determine 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 returned by the driver's call to ddi_dmae_getlim(). (See ddi_dma_lim_x86(9S).) 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 this 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 can participate in a single DMA transfer. ISA bus DMA engines have no scatter/gather capability, so their scatter/gather list sizes are 1. Other finite scatter/gather list sizes would also be possible. For performance reasons, drivers should use the chaining capability if it is available on their parent bus.

As described above, a driver making use of DMA chaining must prefetch DMA cookies before calling ddi_dmae_prog(). The reasons for this are:

- 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 might 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 can 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 for the end of every segment whose number is an integral multiple of the scatter/gather list size to 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 want to use the chaining feature), then the total I/O count for a single DMA operation is 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 for each DMA segment to be 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`).

`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.

`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 performs 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.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	x86

See Also isa(4), attributes(5), ddi_dma_segtocookie(9F), ddi_dmae(9F), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Name ddi_dma_lim_sparc, ddi_dma_lim – SPARC DMA limits structure

Synopsis

```
#include <sys/ddidmareq.h>
```

Interface Level Solaris SPARC DDI specific (Solaris SPARC DDI). These interfaces are obsolete.

Description This page describes the SPARC version of the `ddi_dma_lim` structure. See `ddi_dma_lim_x86(9S)` for a description of the x86 version of this structure.

A `ddi_dma_lim` structure describes in a generic fashion the possible limitations of a device's DMA engine. This information is used by the system when it attempts to set up DMA resources for a device.

Structure Members

```
uint_t  dlim_addr_lo; /* low range of 32 bit
                                       addressing capability */
uint_t  dlim_addr_hi;    /* inclusive upper bound of address.
                             capability */
uint_t  dlim_cntr_max;   /* inclusive upper bound of
                            dma engine address limit * /
uint_t  dlim_burstsizes; /* binary encoded dma burst sizes */
uint_t  dlim_minxfer;    /* minimum effective dma xfer size */
uint_t  dlim_dmaspeed;   /* average dma data rate (kb/s) */
```

The `dlim_addr_lo` and `dlim_addr_hi` fields specify the address range the device's DMA engine can access. The `dlim_addr_lo` field describes the lower 32–bit boundary of the device's DMA engine, the `dlim_addr_hi` describes the inclusive upper 32–bit boundary. The system allocates DMA resources in a way that the address for programming the device's DMA engine (see `ddi_dma_cookie(9S)` or `ddi_dma_htoc(9F)`) is within this range. For example, if your device can access the whole 32–bit address range, you may use [`0,0xFFFFFFFF`]. If your device has just a 16–bit address register but will access the top of the 32–bit address range, then [`0xFFFF0000,0xFFFFFFFF`] is the right limit.

The `dlim_cntr_max` field describes an inclusive upper bound for the device's DMA engine address register. This handles a fairly common case where a portion of the address register is only a latch rather than a full register. For example, the upper 8 bits of a 32–bit address register can be a latch. This splits the address register into a portion that acts as a true address register (24 bits) for a 16 Mbyte segment and a latch (8 bits) to hold a segment number. To describe these limits, specify `0xFFFFFF` in the `dlim_cntr_max` structure.

The `dlim_burstsizes` field describes the possible burst sizes the device's DMA engine can accept. At the time of a DMA resource request, this element defines the possible DMA burst cycle sizes that the requester's DMA engine can handle. The format of the data is binary encoding of burst sizes assumed to be powers of two. That is, if a DMA engine is capable of doing 1–, 2–, 4–, and 16–byte transfers, the encoding ix 0x17. If the device is an SBus device and can take advantage of a 64–bit SBus, the lower 16 bits are used to specify the burst size for 32–bit transfers and the upper 16 bits are used to specify the burst size for 64–bit transfers. As the resource request is handled by the system, the `burstsizes` value can be modified. Prior to enabling DMA for the specific device, the driver that owns the DMA engine should check (using ddi_dma_burstsizes(9F)) what the allowed burstsizes have become and program the DMA engine appropriately.

The dlim_minxfer field describes the minimum effective DMA transfer size (in units of bytes). It must be a power of two. This value specifies the minimum effective granularity of the DMA engine. It is distinct from dlim_burstsizes in that it describes the minimum amount of access a DMA transfer will effect. dlim_burstsizes describes in what electrical fashion the DMA engine might perform its accesses, while dlim_minxfer describes the minimum amount of memory that can be touched by the DMA transfer. As a resource request is handled by the system, the dlim_minxfer value can be modified contingent upon the presence (and use) of I/O caches and DMA write buffers in between the DMA engine and the object that DMA is being performed on. After DMA resources have been allocated, the resultant minimum transfer value can be gotten using ddi_dma_devalign(9F).

The field dlim_dmaspeed is the expected average data rate for the DMA engine (in units of kilobytes per second). Note that this should not be the maximum, or peak, burst data rate, but a reasonable guess as to the average throughput. This field is entirely optional and can be left as zero. Its intended use is to provide some hints about how much of the DMA resource this device might need.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F), ddi_dma_burstsizes(9F), ddi_dma_devalign(9F), ddi_dma_htoc(9F), ddi_dma_setup(9F), ddi_dma_cookie(9S), ddi_dma_lim_x86(9S), ddi_dma_req(9S)

Name ddi_dma_lim_x86 – x86 DMA limits structure

Synopsis

```
#include <sys/ddidmareq.h>
```

Interface Level Solaris x86 DDI specific (Solaris x86 DDI)

Description A `ddi_dma_lim` structure describes in a generic fashion the possible limitations of a device or its DMA engine. This information is used by the system when it attempts to set up DMA resources for a device. When the system is requested to perform a DMA transfer to or from an object, the request is broken up, if necessary, into multiple sub-requests. Each sub–request conforms to the limitations expressed in the `ddi_dma_lim` structure.

This structure should be filled in by calling the routine `ddi_dmae_getlim(9F)`. This routine sets the values of the structure members appropriately based on the characteristics of the DMA engine on the driver's parent bus. If the driver has additional limitations, it can *further restrict* some of the values in the structure members. A driver should *not relax* any restrictions imposed by `ddi_dmae_getlim()`.

Structure Members

```
uint_t  dlim_addr_lo;  /* low range of 32 bit addressing capability */
uint_t  dlim_addr_hi;  /* inclusive upper bound of addressing capability */
uint_t  dlim_minxfer;  /* minimum effective dma transfer size */
uint_t  dlim_version;  /* version number of this structure */
uint_t  dlim_adreg_max; /* inclusive upper bound of
                        /*   incrementing addr reg */
uint_t  dlim_ctreg_max; /* maximum transfer count minus one */
uint_t  dlim_granular; /* granularity (and min size) of transfer count */
short  dlim_sgllen;    /* length of DMA scatter/gather list */
uint_t  dlim_reqsize;  /* maximum transfer size in bytes of a single I/O */
```

The `dlim_addr_lo` and `dlim_addr_hi` fields specify the address range that the device's DMA engine can access. The `dlim_addr_lo` field describes the lower 32–bit boundary of the device's DMA engine. The `dlim_addr_hi` member describes the inclusive, upper 32–bit boundary. The system allocates DMA resources in a way that the address for programming the device's DMA engine will be within this range. For example, if your device can access the whole 32–bit address range, you can use [`0,0xFFFFFFFF`]. See `ddi_dma_cookie(9S)` or `ddi_dma_segtocookie(9F)`.

The `dlim_minxfer` field describes the minimum effective DMA transfer size (in units of bytes), which must be a power of two. This value specifies the minimum effective granularity of the DMA engine and describes the minimum amount of memory that can be touched by the DMA transfer. As a resource request is handled by the system, the `dlim_minxfer` value can be modified. This modification is contingent upon the presence (and use) of I/O caches and DMA write buffers between the DMA engine and the object that DMA is being performed on. After DMA resources have been allocated, you can retrieve the resultant minimum transfer value using `ddi_dma_devalign(9F)`.

The `dlim_version` field specifies the version number of this structure. Set this field to `DMALIM_VER0`.

The dlim_adreg_max field describes an inclusive upper bound for the device's DMA engine address register. This bound handles a fairly common case where a portion of the address register is simply a latch rather than a full register. For example, the upper 16 bits of a 32–bit address register might be a latch. This splits the address register into a portion that acts as a true address register (lower 16 bits) for a 64–kilobyte segment and a latch (upper 16 bits) to hold a segment number. To describe these limits, you specify 0xFFFF in the dlim_adreg_max structure member.

The dlim_ctreg_max field specifies the maximum transfer count that the DMA engine can handle in one segment or cookie. The limit is expressed as the maximum count minus one. This transfer count limitation is a per-segment limitation. Because the limitation is used as a bit mask, it must be one less than a power of two.

The dlim_granular field describes the granularity of the device's DMA transfer ability, in units of bytes. This value is used to specify, for example, the sector size of a mass storage device. DMA requests are broken into multiples of this value. If there is no scatter/gather capability, then the size of each DMA transfer will be a multiple of this value. If there is scatter/gather capability, then a single segment cannot be smaller than the minimum transfer value, but can be less than the granularity. However, the total transfer length of the scatter/gather list is a multiple of the granularity value.

The dlim_sgllen field specifies the maximum number of entries in the scatter/gather list. This value is the number of segments or cookies that the DMA engine can consume in one I/O request to the device. If the DMA engine has no scatter/gather list, set this field to one.

The dlim_reqsize field describes the maximum number of bytes that the DMA engine can transmit or receive in one I/O command. This limitation is only significant if it is less than (dlim_ctreg_max +1) * dlim_sgllen. If the DMA engine has no particular limitation, set this field to 0xFFFFFFFF.

See Also ddi_dmae(9F), ddi_dma_addr_setup(9F), ddi_dma_buf_setup(9F), ddi_dma_devalign(9F), ddi_dma_segtocookie(9F), ddi_dma_setup(9F), ddi_dma_cookie(9S) ddi_dma_lim_sparc(9S), ddi_dma_req(9S)

Name ddi_dma_req – DMA Request structure

Synopsis `#include <sys/ddidmareq.h>`

Interface Level Solaris DDI specific (Solaris DDI). This interface is obsolete.

Description A `ddi_dma_req` structure describes a request for DMA resources. A driver can use it to describe forms of allocations and ways to allocate DMA resources for a DMA request.

Structure Members

```
ddi_dma_lim_t  *dmar_limits;       /* Caller's dma engine
                                      constraints */
uint_t         dmar_flags;         /* Contains info for
                                      mapping routines */
int            (*dmar_fp)(caddr_t);/* Callback function */
caddr_t        dmar_arg;           /* Callback function's argument */
ddi_dma_obj_t  dmar_object;        /* Descrip. of  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 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 might 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 might not have the hardware ability to support it.

`DDI_DMA_PARTIAL` lets the system know that the caller can accept partial mapping. That is, if the size of the object exceeds the resources available, the system allocates only a portion of the object and returns status indicating this partial allocation. 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 an 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.

Many machine implementations make this non-sequential memory access 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 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 can create a condition in which a system will pick resources in a way that makes these synchronization steps are as efficient as possible.

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

The callback function specified by the member `dmar_fp` indicates how a caller to one of the DMA resource allocation functions wants to deal with the possibility of resources not being available. (See `ddi_dma_setup`(9F).) 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. Setting `dmar_fp` to `DDI_DMA_SLEEP` indicates the caller wants to have 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 later, when resources 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 resource but failed to do so, again, in which case the callback function will be put back on a list to be called again later.

`1` Indicating either success at allocating DMA resources or that it no longer wants to retry.

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

It is possible that a call to `ddi_dma_free`(9F), which frees DMA resources, might cause a callback function to be called and, unless some care is taken, an undesired recursion can occur. This can cause an undesired recursive `mutex_enter`(9F), which makes the system panic.

dmar_object Structure

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

```
uint_t            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 allocated for DMA.

The dmao_type element selects the kind of object described by dmao_obj. It can 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 */
      void         *v_priv;  /* priv data for shadow I/O */
};
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

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

Name ddi_fm_error – I/O error status structure

Synopsis

```
#include <sys/ddifm.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description A ddi_fm_error_t structure contains common data necessary for I/O error handling. A pointer to a ddi_fm_error_t structure is passed to error handling callbacks where it can then be used in a call to pci_ereport_post(). The same structure is also returned to callers of ddi_fm_acc_err_get() and ddi_fm_dma_err_get().

Structure Members

```
int                 fme_version;
uint64_t            fme_ena;
int                 fme_status;
int                 fme_flag;
ddi_acc_handle_t    fme_acc_handle;
ddi_dma_handle_t    fme_dma_handle;
```

The fme_version is the current version of ddi_fm_error_t. Valid values for the version are: DDI_FME_VER0 and DDI_FME_VER1.

The fme_ena is the FMA event protocol Format 1 Error Numeric Association (ENA) for this error condition.

The fme_flag field is set to DDI_FM_ERR_EXPECTED if the error is the result of a DDI_ACC_CAUTIOUS protected operation. In this case, fme_acc_handle is valid and the driver should check for and report only errors not associated with the DDI_ACC_CAUTIOUS protected access operation. This field can also be set to DDI_FM_ERR_POKE or DDI_FM_ERR_PEEK if the error is the result of a ddi_peek(9F) or ddi_poke(9F) operation. The driver should handle these in a similar way to DDI_FM_ERR_EXPECTED. Otherwise, ddi_flag is set to DDI_FM_ERR_UNEXPECTED and the driver must perform the full range of error handling tasks.

The fme_status indicates current status of an error handler callback or resource handle:

DDI_FM_OK
: No errors were detected.

DDI_FM_FATAL
: An error which is considered fatal to the operational state of the system was detected.

DDI_FM_NONFATAL
: An error which is not considered fatal to the operational state of the system was detected.

DDI_FM_UNKNOWN
: An error was detected, but the driver was unable to determine the impact of the error on the operational state of the system.

The fme_acc_handle is the valid access handle associated with the error that can be returned from pci_ereport_post()

The fme_dma_handle is the valid DMA handle associated with the error that can be returned from pci_ereport_post()

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also attributes(5), ddi_fm_acc_err_get(9F), ddi_fm_dma_err_get(9F), ddi_fm_handler_register(9F), ddi_peek(9F), ddi_poke(9F), pci_ereport_post(9F)

Writing Device Drivers

Name ddi-forceattach, ddi-no-autodetach – properties controlling driver attach/detach behavior

Description Solaris device drivers are attached by `devfsadm`(1M) and by the kernel in response to `open`(2) requests from applications. Drivers not currently in use can be detached when the system experiences memory pressure. The `ddi-forceattach` and `ddi-no-autodetach` properties can be used to customize driver attach/detach behavior.

The `ddi-forceattach` is an integer property, to be set globally by means of the `driver.conf`(4) file. Drivers with this property set to 1 are loaded and attached to all possible instances during system startup. The driver will not be auto-detached due to system memory pressure.

The `ddi-no-autodetach` is an integer property to be set globally by means of the `driver.conf`(4) file or created dynamically by the driver on a per-instance basis with `ddi_prop_update_int`(9F). When this property is set to 1, the kernel will not auto-detach driver due to system memory pressure.

Note that `ddi-forceattach` implies `ddi-no-autodetach`. Setting either property to a non-integer value or an integer value not equal to 1 produces undefined results. These properties do not prevent driver detaching in response to reconfiguration requests, such as executing commands `cfgadm`(1M), `modunload`(1M), `rem_drv`(1M), and `update_drv`(1M).

See Also `driver.conf`(4)

Writing Device Drivers

Name ddi_idevice_cookie – device interrupt cookie

Synopsis

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI). This interface is obsolete. Use the new interrupt interfaces referenced in `Intro`(9F). Refer to *Writing Device Drivers* for more information.

Description The `ddi_idevice_cookie_t` structure contains interrupt priority and interrupt vector information for a device. This structure is useful for devices having programmable bus-interrupt levels. `ddi_add_intr`(9F) assigns values to the `ddi_idevice_cookie_t` structure members.

Structure Members

```
u_short    idev_vector;      /* interrupt vector */
ushort_t    idev_priority;    /* interrupt priority */
```

The `idev_vector` field contains the interrupt vector number for vectored bus architectures such as VMEbus. The `idev_priority` field contains the bus interrupt priority level.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

See Also `ddi_add_intr`(9F), `Intro`(9F)

Writing Device Drivers

Name devmap_callback_ctl – device mapping-control structure

Synopsis

```
#include <sys/ddidevmap.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description A devmap_callback_ctl structure describes a set of callback routines that are called by the system to notify a device driver to manage events on the device mappings created by devmap_setup(9F) or ddi_devmap_segmap(9F).

Device drivers pass the initialized devmap_callback_ctl structure to either devmap_devmem_setup(9F) or devmap_umem_setup(9F) in the devmap(9E) entry point during the mapping setup. The system makes a private copy of the structure for later use. Device drivers can specify different devmap_callback_ctl for different mappings.

A device driver should allocate the device mapping control structure and initialize the following fields, if the driver wants the entry points to be called by the system:

devmap_rev
: Version number. Set this to DEVMAP_OPS_REV.

devmap_map
: Set to the address of the devmap_map(9E) entry point or to NULL if the driver does not support this callback. If set, the system calls the devmap_map(9E) entry point during the mmap(2) system call. The drivers typically allocate driver private data structure in this function and return the pointer to the private data structure to the system for later use.

devmap_access
: Set to the address of the devmap_access(9E) entry point or to NULL if the driver does not support this callback. If set, the system calls the driver's devmap_access(9E) entry point during memory access. The system expects devmap_access(9E) to call either devmap_do_ctxmgt(9F) or devmap_default_access(9F) to load the memory address translations before it returns to the system.

devmap_dup
: Set to the address of the devmap_dup(9E) entry point or to NULL if the driver does not support this call. If set, the system calls the devmap_dup(9E) entry point during the fork(2) system call.

devmap_unmap
: Set to the address of the devmap_unmap(9E) entry point or to NULL if the driver does not support this call. If set, the system will call the devmap_unmap(9E) entry point during the munmap(2) or exit(2) system calls.

Structure Members

```
int    devmap_rev;
int    (*devmap_map)(devmap_cookie_t dhp, dev_t dev,
     uint_t flags,offset_t off, size_t len, void **pvtp);
int    (*devmap_access)(devmap_cookie_t dhp, void *pvtp,
     offset_t off, size_t len, uint_t type, uint_t rw);
int    (*devmap_dup)(devmap_cookie_t dhp, void *pvtp,
           devmap_cookie_t new_dhp, void **new_pvtp);
```

```
void    (*devmap_unmap)(devmap_cookie_t dhp, void *pvtp,
     offset_t off, size_t len, devmap_cookie_t new_dhp1,
     void **new_pvtp1, devmap_cookie_t new_dhp2, void **new_pvtp2);
```

See Also exit(2), fork(2), mmap(2), munmap(2), devmap(9E), devmap_access(9E), devmap_dup(9E), devmap_map(9E), devmap_unmap(9E), ddi_devmap_segmap(9F), devmap_default_access(9F), devmap_devmem_setup(9F), devmap_do_ctxmgt(9F), devmap_setup(9F), devmap_umem_setup(9F)

Writing Device Drivers

Name dev_ops – device operations structure

Synopsis

```
#include <sys/conf.h>
#include <sys/devops.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description dev_ops contains driver common fields and pointers to the bus_ops and cb_ops(9S).

Following are the device functions provided in the device operations structure. All fields must be set at compile time.

devo_rev	Driver build version. Set this to DEVO_REV.
devo_refcnt	Driver reference count. Set this to 0.
devo_getinfo	Get device driver information (see getinfo(9E)).
devo_identify	This entry point is obsolete. Set to nulldev.
devo_probe	Probe device. See probe(9E).
devo_attach	Attach driver to dev_info. See attach(9E).
devo_detach	Detach/prepare driver to unload. See detach(9E).
devo_reset	Reset device. (Not supported in this release.) Set this to nodev.
devo_cb_ops	Pointer to cb_ops(9S) structure for leaf drivers.
devo_bus_ops	Pointer to bus operations structure for nexus drivers. Set this to NULL if this is for a leaf driver.
devo_power	Power a device attached to system. See power(9E).

Structure Members

```
int                devo_rev;
int                devo_refcnt;
int                (*devo_getinfo)(dev_info_t *dip,
                   ddi_info_cmd_t infocmd, void *arg, void **result);
int                (*devo_identify)(dev_info_t *dip);
int                (*devo_probe)(dev_info_t *dip);
int                (*devo_attach)(dev_info_t *dip,
                   ddi_attach_cmd_t cmd);
int                (*devo_detach)(dev_info_t *dip,
                   ddi_detach_cmd_t cmd);
int                (*devo_reset)(dev_info_t *dip, ddi_reset_cmd_t cmd);
struct cb_ops      *devo_cb_ops;
struct bus_ops     *devo_bus_ops;
int                (*devo_power)(dev_info_t *dip, int component, int level);
```

See Also attach(9E), detach(9E), getinfo(9E), probe(9E), power(9E), nodev(9F)

Writing Device Drivers

Name fmodsw – STREAMS module declaration structure

Synopsis

```
#include <sys/stream.h>
#include <sys/conf.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `fmodsw` structure contains information for STREAMS modules. All STREAMS modules must define a `fmodsw` structure.

`f_name` must match `mi_idname` in the `module_info` structure. See `module_info`(9S). `f_name` should also match the module binary name. (See WARNINGS.)

All modules must set the `f_flag` to `D_MP` to indicate that they safely allow multiple threads of execution. See `mt-streams`(9F) for additional flags.

Structure Members

```
char              f_name[FMNAMESZ + 1];   /* module name */
struct streamtab  *f_str;                 /* streams information */
int               f_flag;                 /* flags */
```

See Also `mt-streams`(9F), `modlstrmod`(9S), `module_info`(9S)

STREAMS Programming Guide

Warnings If `f_name` does not match the module binary name, unexpected failures can occur.

Name free_rtn – structure that specifies a driver's message-freeing routine

Synopsis `#include <sys/stream.h>`

Interface Level Architecture independent level 1 (DDI/DKI).

Description The `free_rtn` structure is referenced by the `datab` structure. When `freeb(9F)` is called to free the message, the driver's message-freeing routine (referenced through the `free_rtn` structure) is called, with arguments, to free the data buffer.

Structure Members

```
void     (*free_func)()      /* user's freeing routine */
char     *free_arg           /* arguments to free_func() */
```

The `free_rtn` structure is defined as type `frtn_t`.

See Also `esballoc`(9F), `freeb`(9F), `datab`(9S)

STREAMS Programming Guide

Name gld_mac_info – Generic LAN Driver MAC info data structure

Synopsis

```
#include <sys/gld.h>
```

Interface Level Solaris architecture specific (Solaris DDI).

Description The Generic LAN Driver (GLD) Media Access Control (MAC) information (`gld_mac_info`) structure is the main data interface between the device-specific driver and GLD. It contains data required by GLD and a pointer to an optional additional driver-specific information structure.

The `gld_mac_info` structure should be allocated using `gld_mac_alloc()` and deallocated using `gld_mac_free()`. Drivers can make no assumptions about the length of this structure, which might be different in different releases of Solaris and/or GLD. Structure members private to GLD, not documented here, should not be set or read by the device-specific driver.

Structure Members

```
caddr_t         gldm_private;           /* Driver private data */
int               (*gldm_reset)();         /* Reset device */
int               (*gldm_start)();         /* Start device */
int               (*gldm_stop)();          /* Stop device */
int               (*gldm_set_mac_addr)();  /* Set device phys addr */
int               (*gldm_set_multicast)(); /* Set/delete */
                                           /* multicast address */
int               (*gldm_set_promiscuous)();/* Set/reset */
                                           /* promiscuous mode */
int               (*gldm_send)();          /* Transmit routine */
u_int             (*gldm_intr)();          /* Interrupt handler */
int               (*gldm_get_stats)();     /* Get device statistics */
int               (*gldm_ioctl)();         /* Driver-specific ioctls */
char             *gldm_ident;              /* Driver identity string */
uint32_t          gldm_type;               /* Device type */
uint32_t          gldm_minpkt;             /* Minimum packet size */
                                           /* accepted by driver */
uint32_t          gldm_maxpkt;             /* Maximum packet size */
                                           /* accepted by driver */
uint32_t          gldm_addrlen;            /* Physical address */
                                           /* length */
int32_t           gldm_saplen;             /* SAP length for */
                                           /* DL_INFO_ACK */
unsigned char    *gldm_broadcast_addr;     /* Physical broadcast */
                                           /* addr */
unsigned char    *gldm_vendor_addr;        /* Factory MAC address */
t_uscalar_t       gldm_ppa;                /* Physical Point of */
                                           /* Attachment (PPA) number */
dev_info_t       *gldm_devinfo;            /* Pointer to device's */
                                           /* dev_info node */
ddi_iblock_cookie_tgldm_cookie;            /* Device's interrupt */
                                           /* block cookie */
```

```
uint32_t         gldm_capabilities;        /* Device capabilities */
```

Below is a description of the members of the `gld_mac_info` structure that are visible to the device driver.

`gldm_private`	This structure member is private to the device-specific driver and is not used or modified by GLD. Conventionally, this is used as a pointer to private data, pointing to a driver-defined and driver-allocated per-instance data structure.

The following group of structure members must be set by the driver before calling `gld_register()`, and should not thereafter be modified by the driver; `gld_register()` can use or cache the values of some of these structure members, so changes made by the driver after calling `gld_register()` might cause unpredicted results.

`gldm_reset`	Pointer to driver entry point; see `gld`(9E).
`gldm_start`	Pointer to driver entry point; see `gld`(9E).
`gldm_stop`	Pointer to driver entry point; see `gld`(9E).
`gldm_set_mac_addr`	Pointer to driver entry point; see `gld`(9E).
`gldm_set_multicast`	Pointer to driver entry point; see `gld`(9E).
`gldm_set_promiscuous`	Pointer to driver entry point; see `gld`(9E).
`gldm_send`	Pointer to driver entry point; see `gld`(9E).
`gldm_intr`	Pointer to driver entry point; see `gld`(9E).
`gldm_get_stats`	Pointer to driver entry point; see `gld`(9E).
`gldm_ioctl`	Pointer to driver entry point; can be `NULL`; see `gld`(9E).
`gldm_ident`	Pointer to a string containing a short description of the device. It is used to identify the device in system messages.
`gldm_type`	The type of device the driver handles. The values currently supported by GLD are `DL_ETHER` (IEEE 802.3 and Ethernet Bus), `DL_TPR` (IEEE 802.5 Token Passing Ring), and `DL_FDDI` (ISO 9314-2 Fibre Distributed Data Interface). This structure member must be correctly set for GLD to function properly. **Note –** Support for the DL_TPR and DL_FDDI media types is obsolete and may be removed in a future release of Solaris.
`gldm_minpkt`	Minimum *Service Data Unit* size — the minimum packet size, not including the MAC header, that the device will transmit. This can be zero if the device-specific driver can handle any required padding.

gldm_maxpkt — Maximum *Service Data Unit* size — the maximum size of packet, not including the MAC header, that can be transmitted by the device. For Ethernet, this number is 1500.

gldm_addrlen — The length in bytes of physical addresses handled by the device. For Ethernet, Token Ring, and FDDI, the value of this structure member should be 6.

gldm_saplen — The length in bytes of the Service Access Point (SAP) address used by the driver. For GLD-based drivers, this should always be set to -2, to indicate that two-byte SAP values are supported and that the SAP appears *after* the physical address in a DLSAP address. See the description under "Message DL_INFO_ACK" in the DLPI specification for more details.

gldm_broadcast_addr — Pointer to an array of bytes of length gldm_addrlen containing the broadcast address to be used for transmit. The driver must allocate space to hold the broadcast address, fill it in with the appropriate value, and set gldm_broadcast_addr to point at it. For Ethernet, Token Ring, and FDDI, the broadcast address is normally 0xFF-FF-FF-FF-FF-FF.

gldm_vendor_addr — Pointer to an array of bytes of length gldm_addrlen containing the vendor-provided network physical address of the device. The driver must allocate space to hold the address, fill it in with information read from the device, and set gldm_vendor_addr to point at it.

gldm_ppa — The Physical Point of Attachment (PPA) number for this instance of the device. Normally this should be set to the instance number, returned from ddi_get_instance(9F).

gldm_devinfo — Pointer to the dev_info node for this device.

gldm_cookie — The interrupt block cookie returned by ddi_get_iblock_cookie(9F), ddi_add_intr(9F), ddi_get_soft_iblock_cookie(9F), or ddi_add_softintr(9F). This must correspond to the device's receive interrupt, from which gld_recv() is called.

gldm_capabilities — Bit-field of device capabilities. If the device is capable of reporting media link state, the GLD_CAP_LINKSTATE bit should be set.

See Also gld(7D), dlpi(7P), attach(9E), gld(9E), ddi_add_intr(9F), gld(9F), gld_stats(9S)

Writing Device Drivers

Name gld_stats – Generic LAN Driver statistics data structure

Synopsis `#include <sys/gld.h>`

Interface Level Solaris architecture specific (Solaris DDI).

Description The Generic LAN Driver (GLD) statistics (`gld_stats`) structure is used to communicate statistics and state information from a GLD-based driver to GLD when returning from a driver's `gldm_get_stats()` routine as discussed in `gld`(9E) and `gld`(7D). The members of this structure, filled in by the GLD-based driver, are used when GLD reports the statistics. In the tables below, the name of the statistics variable reported by GLD is noted in the comments. See `gld`(7D) for a more detailed description of the meaning of each statistic.

Drivers can make no assumptions about the length of this structure, which might be different in different releases of Solaris and/or GLD. Structure members private to GLD, not documented here, should not be set or read by the device specific driver.

Structure Members The following structure members are defined for all media types:

```
uint64_t   glds_speed;                /* ifspeed */
uint32_t   glds_media;                /* media */
uint32_t   glds_intr;                 /* intr */
uint32_t   glds_norcvbuf;             /* norcvbuf */
uint32_t   glds_errrcv;               /* ierrors */
uint32_t   glds_errxmt;               /* oerrors */
uint32_t   glds_missed;               /* missed */
uint32_t   glds_underflow;            /* uflo */
uint32_t   glds_overflow;             /* oflo */
```

The following structure members are defined for media type `DL_ETHER`:

```
uint32_t   glds_frame;                    /* align_errors */
uint32_t   glds_crc;                      /* fcs_errors */
uint32_t   glds_duplex;                   /* duplex */
uint32_t   glds_nocarrier;                /* carrier_errors */
uint32_t   glds_collisions;               /* collisions */
uint32_t   glds_excoll;                   /* ex_collisions */
uint32_t   glds_xmtlatecoll;              /* tx_late_collisions */
uint32_t   glds_defer;                    /* defer_xmts */
uint32_t   glds_dot3_first_coll;          /* first_collisions */
uint32_t   glds_dot3_multi_coll;          /* multi_collisions */
uint32_t   glds_dot3_sqe_error;           /* sqe_errors */
uint32_t   glds_dot3_mac_xmt_error;       /* macxmt_errors */
uint32_t   glds_dot3_mac_rcv_error;       /* macrcv_errors */
uint32_t   glds_dot3_frame_too_long;      /* toolong_errors */
uint32_t   glds_short;                    /* runt_errors */
```

The following structure members are defined for media type `DL_TPR`:

```
uint32_t   glds_dot5_line_error          /* line_errors */
uint32_t   glds_dot5_burst_error         /* burst_errors */
uint32_t   glds_dot5_signal_loss         /* signal_losses */
uint32_t   glds_dot5_ace_error           /* ace_errors */
uint32_t   glds_dot5_internal_error      /* internal_errors */
uint32_t   glds_dot5_lost_frame_error    /* lost_frame_errors */
uint32_t   glds_dot5_frame_copied_error  /* frame_copied_errors */
uint32_t   glds_dot5_token_error         /* token_errors */
uint32_t   glds_dot5_freq_error          /* freq_errors */
```

Note – Support for the DL_TPR media type is obsolete and may be removed in a future release of Solaris.

The following structure members are defined for media type DL_FDDI:

```
uint32_t   glds_fddi_mac_error;          /* mac_errors */
uint32_t   glds_fddi_mac_lost;           /* mac_lost_errors */
uint32_t   glds_fddi_mac_token;          /* mac_tokens */
uint32_t   glds_fddi_mac_tvx_expired;    /* mac_tvx_expired */
uint32_t   glds_fddi_mac_late;           /* mac_late */
uint32_t   glds_fddi_mac_ring_op;        /* mac_ring_ops */
```

Note – Support for the DL_FDDI media type is obsolete and may be removed in a future release of Solaris.

Most of the above statistics variables are counters denoting the number of times the particular event was observed. Exceptions are:

glds_speed
: An estimate of the interface's current bandwidth in bits per second. For interfaces that do not vary in bandwidth or for those where no accurate estimation can be made, this object should contain the nominal bandwidth.

glds_media
: The type of media (wiring) or connector used by the hardware. Currently supported media names include GLDM_AUI, GLDM_BNC, GLDM_TP, GLDM_10BT, GLDM_100BT, GLDM_100BTX, GLDM_100BT4, GLDM_RING4, GLDM_RING16, GLDM_FIBER, and GLDM_PHYMII. GLDM_UNKNOWN can also be specified.

glds_duplex
: Current duplex state of the interface. Supported values are GLD_DUPLEX_HALF and GLD_DUPLEX_FULL. GLD_DUPLEX_UNKNOWN can also be specified.

See Also

gld(7D), gld(9F), gld(9E), gld_mac_info(9S)

Writing Device Drivers

Name hook_nic_event – data structure describing events related to network interfaces

Synopsis

```
#include <sys/neti.h>
#include <sys/hook.h>
#include <sys/hook_event.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The `hook_nic_event` structure contains fields that relate to an event that has occurred and belongs to a network interface. This structure is passed through to callbacks for `NE_PLUMB`, `NE_UNPLUMB`, `NE_UP`, `NE_DOWN` and `NE_ADDRESS_CHANGE` events.

A callback may not alter any of the fields in this structure.

Structure Members

```
net_data_t          hne_family;
phy_if_t            pkt_private;
lif_if_t            hne_lif;
nic_event_t         hne_event;
nic_event_data_t    hne_data;
size_t              hne_datalen;
```

The following fields are set for each event:

`hne_family` A valid reference for the network protocol that owns this network interface and can be in calls to other `netinfo(9F)` functions.

`hne_nic` The physical interface to which an event belongs.

`hne_event` A value that indicates the respective event. The current list of available events is:

`NE_PLUMB`
: an interface has just been created.

`NE_UNPLUMB`
: An interface has just been destroyed and no more events should be received for it.

`NE_UP`
: An interface has changed the state to “up” and may now generate packet events.

`NE_DOWN`
: An interface has changed the state to “down” and will no longer generate packet events.

`NE_ADDRESS_CHANGE`
An address on an interface has changed. `hne_lif` refers to the logical interface for which the change is occurring, `hne_data` is a pointer to a `sockaddr` structure that is `hne_datalen` bytes long and contains the new network address.

`NE_IFINDEX_CHANGE`
An interface index has changed. `hne_lif` refers to the logical interface for which the change is occurring, `hne_data` is a new *ifindex* value.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `attributes(5)`, `netinfo(9F)`

Name hook_pkt_event – packet event structure passed through to hooks

Synopsis

```
#include <sys/neti.h>
#include <sys/hook.h>
#include <sys/hook_event.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The `hook_pkt_event` structure contains fields that relate to a packet in a network protocol handler. This structure is passed through to a callback for `NH_PRE_ROUTING`, `NH_POST_ROUTING`, `NH_FORWARDING`, `NH_LOOPBACK_IN` and `NH_LOOPBACK_OUT` events.

A callback may only modify the `hpe_hdr`, `hpe_mp` and `hpe_mb` fields.

The following table documents which fields can be safely used as a result of each event.

Event	hpe_ifp	hpe_ofp	hpe_hdr	hpe_mp	hpe_mb
NH_PRE_ROUTING	yes		yes	yes	yes
NH_POST_ROUTING		yes	yes	yes	yes
NH_FORWARDING	yes	yes	yes	yes	yes
NH_LOOPBACK_IN	yes		yes	yes	yes
NH_LOOPBACK_OUT		yes	yes	yes	yes

Structure Members

```
net_data_t          hne_family;
phy_if_t            hpe_ifp;
phy_if_t            hpe_ofp;
void                *hpe_hdr;
mblk_t              *hpe_mp;
mblk_t              *hpe_mb;
uint32_t            hpe_flags;
```

The following fields are set for each event:

`hne_family` The protocol family for this packet. This value matches the corresponding value returned from a call to `net_protocol_lookup`(9F).

`hpe_ifp` The inbound interface for a packet.

`hpe_ofp` The outbound interface for a packet.

`hpe_hdr` Pointer to the start of the network protocol header within an `mblk_t` structure.

`hpe_mp` Pointer to the `mblk_t` pointer that points to the first `mblk_t` structure in this packet.

`hpe_mb` Pointer to the `mblk_t` structure that contains `hpe_hdr`.

`hpe_flags` This field is used to carry additional properties of packets. The current collection of defined bits available is:

HPE_BROADCAST — This bit is set if the packet was recognized as a broadcast packet from the link layer. The bit cannot be set if HPE_MULTICAST is set, currently only possible with physical in packet events.

HPE_MULTICAST — This set if the packet was recognized as a multicast packet from the link layer. This bit cannot be set if HPE_BROADCAST is set, currently only possible with physical in packet events.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also net_protocol_lookup(9F), netinfo(9F)

Name hook_t – callback structure for subscribing to netinfo events

Synopsis

```
#include <sys/hook.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The `hook_t` data structure defines a callback that is to be inserted into a networking event. This data structure must be allocated with a call to `hook_alloc()` and released with a call to `hook_free()`.

Structure Members

```
hook_func_t  h_func;      /* callback function to invoke */
char         *h_name;     /* unique name given to the hook */
int          h_flags;
hook_hint_t  h_hint;      /* insertion hint type */
uintptr_t    h_hintvalue; /* used with h_hint */
void         *h_arg;      /* value to pass into h_func */

typedef int (*hook_func_t)(net_event_t token, hook_data_t info,
              void *);
```

HINT TYPES Hook hints are hints that are used at the time of insertion and are not rules that enforce where a hook lives for its entire lifetime on an event. The valid values for the `h_hint` field are:

`HH_NONE` Insert the hook wherever convenient.

`HH_FIRST` Place the hook first on the list of hooks.

`HH_LAST` Place the hook last on the list of hooks.

`HH_BEFORE` Place the hook before another hook on the list of hooks. The value in `h_hintvalue` must be a pointer to the name of another hook.

`HH_AFTER` Place the hook after another hook on the list of hooks. The value in `h_hintvalue` must be a pointer to the name of another hook.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `netinfo(9F)`

Name inquiry-device-type, inquiry-vendor-id, inquiry-product-id, inquiry-revision-id – properties from SCSI inquiry data

Description These are optional properties created by the system for SCSI target devices.

`inquiry-device-type` is an integer property. When present, the least significant byte of the value indicates the device type as defined by the SCSI standard.

`inquiry-vendor-id` is a string property. When present, it contains the SCSI vendor identification inquiry data (from SCSI inquiry data bytes 8 - 15), formatted as a NULL-terminated string.

`inquiry-product-id` is a string property. When present, it contains the SCSI product identification inquiry data (from SCSI inquiry data bytes 16 - 31).

`inquiry-revision-id` is a string property. When present, it contains the SCSI product revision inquiry data (from SCSI inquiry data bytes 32 - 35).

Consumers of these properties should compare the property values with `DTYPE_*` values defined in `<sys/scsi/generic/inquiry.h>`.

See Also *Writing Device Drivers*

Name iocblk – STREAMS data structure for the M_IOCTL message type

Synopsis

```
#include <sys/stream.h>
```

Interface Level Architecture independent level 1 (DDI/DKI).

Description The `iocblk` data structure is used for passing `M_IOCTL` messages.

Structure Members

```
int         ioc_cmd;      /* ioctl command type */
cred_t      *ioc_cr;      /* full credentials */
uint_t      ioc_id;       /* ioctl id */
uint_t      ioc_flag;     /* ioctl flags */
uint_t      ioc_count;    /* count of bytes in data field */
int         ioc_rval;     /* return value */
int         ioc_error;    /* error code */
```

See Also *STREAMS Programming Guide*

Name iovec – data storage structure for I/O using uio

Synopsis

```
#include <sys/uio.h>
```

Interface Level Architecture independent level 1 (DDI/DKI).

Description An `iovec` structure describes a data storage area for transfer in a `uio(9S)` structure. Conceptually, it can be thought of as a base address and length specification.

Structure Members

```
caddr_t    iov_base;  /* base address of the data storage area */
                      /* represented by the iovec structure */
int        iov_len;   /* size of the data storage area in bytes */
```

See Also `uio(9S)`

Writing Device Drivers

Name kstat – kernel statistics structure

Synopsis

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description Each kernel statistic (kstat) exported by device drivers consists of a header section and a data section. The kstat structure is the header portion of the statistic.

A driver receives a pointer to a kstat structure from a successful call to kstat_create(9F). Drivers should never allocate a kstat structure in any other manner.

After allocation, the driver should perform any further initialization needed before calling kstat_install(9F) to actually export the kstat.

Structure Members

```
void      *ks_data;              /* kstat type-specif. data */
ulong_t   ks_ndata;              /* # of type-specif. data
                                     records */
ulong_t   ks_data_size;          /* total size of kstat data
                                    section */
int       (*ks_update)(struct kstat *, int);
void      *ks_private;           /* arbitrary provider-private
                                    data */
void      *ks_lock;              /* protects kstat's data */
```

The members of the kstat structure available to examine or set by a driver are as follows:

ks_data — Points to the data portion of the kstat. Either allocated by kstat_create(9F) for the drivers use, or by the driver if it is using virtual kstats.

ks_ndata — The number of data records in this kstat. Set by the ks_update(9E) routine.

ks_data_size — The amount of data pointed to by ks_data. Set by the ks_update(9E) routine.

ks_update — Pointer to a routine that dynamically updates kstat. This is useful for drivers where the underlying device keeps cheap hardware statistics, but where extraction is expensive. Instead of constantly keeping the kstat data section up to date, the driver can supply a ks_update(9E) function that updates the kstat data section on demand. To take advantage of this feature, set the ks_update field before calling kstat_install(9F).

ks_private — Is a private field for the driver's use. Often used in ks_update(9E).

`ks_lock` — Is a pointer to a mutex that protects this `kstat`. `kstat` data sections are optionally protected by the per-`kstat` `ks_lock`. If `ks_lock` is non-`NULL`, `kstat` clients (such as `/dev/kstat`) will acquire this lock for all of their operations on that `kstat`. It is up to the `kstat` provider to decide whether guaranteeing consistent data to `kstat` clients is sufficiently important to justify the locking cost. Note, however, that most statistic updates already occur under one of the provider's mutexes. If the provider sets `ks_lock` to point to that mutex, then `kstat` data locking is free. `ks_lock` is really of type `(kmutex_t*)` and is declared as `(void*)` in the `kstat` header. That way, users do not have to be exposed to all of the kernel's lock-related data structures.

See Also `kstat_create(9F)`

Writing Device Drivers

Name kstat_intr – structure for interrupt kstats

Synopsis

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description Interrupt statistics are kept in the `kstat_intr` structure. When `kstat_create(9F)` creates an interrupt `kstat`, the `ks_data` field is a pointer to one of these structures. The macro `KSTAT_INTR_PTR()` is provided to retrieve this field. It looks like this:

```
#define KSTAT_INTR_PTR(kptr)    ((kstat_intr_t *)(kptr)->ks_data)
```

An interrupt is a hard interrupt (sourced from the hardware device itself), a soft interrupt (induced by the system through the use of some system interrupt source), a watchdog interrupt (induced by a periodic timer call), spurious (an interrupt entry point was entered but there was no interrupt to service), or multiple service (an interrupt was detected and serviced just prior to returning from any of the other types).

Drivers generally report only claimed hard interrupts and soft interrupts from their handlers, but measurement of the spurious class of interrupts is useful for auto-vectored devices in order to pinpoint any interrupt latency problems in a particular system configuration.

Devices that have more than one interrupt of the same type should use multiple structures.

Structure Members

```
ulong_t    intrs[KSTAT_NUM_INTRS];    /* interrupt counters */
```

The only member exposed to drivers is the `intrs` member. This field is an array of counters. The driver must use the appropriate counter in the array based on the type of interrupt condition.

The following indexes are supported:

`KSTAT_INTR_HARD`	Hard interrupt
`KSTAT_INTR_SOFT`	Soft interrupt
`KSTAT_INTR_WATCHDOG`	Watchdog interrupt
`KSTAT_INTR_SPURIOUS`	Spurious interrupt
`KSTAT_INTR_MULTSVC`	Multiple service interrupt

See Also `kstat(9S)`

Writing Device Drivers

Name kstat_io – structure for I/O kstats

Synopsis

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description I/O kstat statistics are kept in a `kstat_io` structure. When `kstat_create(9F)` creates an I/O `kstat`, the `ks_data` field is a pointer to one of these structures. The macro `KSTAT_IO_PTR()` is provided to retrieve this field. It looks like this:

```
#define KSTAT_IO_PTR(kptr)    ((kstat_io_t *)(kptr)->ks_data)
```

Structure Members

```
u_longlong_t      nread;      /* number of bytes read */
u_longlong_t      nwritten;   /* number of bytes written *]/
ulong_t           reads;      /* number of read operations */
ulong_t           writes;     /* number of write operations */
```

The `nread` field should be updated by the driver with the number of bytes successfully read upon completion.

The `nwritten` field should be updated by the driver with the number of bytes successfully written upon completion.

The `reads` field should be updated by the driver after each successful read operation.

The `writes` field should be updated by the driver after each successful write operation.

Other I/O statistics are updated through the use of the `kstat_queue(9F)` functions.

See Also `kstat_create(9F)`, `kstat_named_init(9F)`, `kstat_queue(9F)`, `kstat_runq_back_to_waitq(9F)`, `kstat_runq_enter(9F)`, `kstat_runq_exit(9F)`, `kstat_waitq_enter(9F)`, `kstat_waitq_exit(9F)`, `kstat_waitq_to_runq(9F)`

Writing Device Drivers

Name kstat_named – structure for named kstats

Synopsis

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description Named `kstats` are an array of name-value pairs. These pairs are kept in the `kstat_named` structure. When a `kstat` is created by `kstat_create(9F)`, the driver specifies how many of these structures will be allocated. The structures are returned as an array pointed to by the `ks_data` field.

Structure Members

```
union {
          char                  c[16];
          long                  l;
          ulong_t               ul;
          longlong_t            ll;
          u_longlong_t          ull;
} value;  /* value of counter */
```

The only member exposed to drivers is the `value` member. This field is a union of several data types. The driver must specify which type it will use in the call to `kstat_named_init()`.

See Also `kstat_create(9F)`, `kstat_named_init(9F)`

Writing Device Drivers

Name linkblk – STREAMS data structure sent to multiplexor drivers to indicate a link

Synopsis

```
#include <sys/stream.h>
```

Interface Level Architecture independent level 1 (DDI/DKI)

Description The `linkblk` structure is used to connect a lower Stream to an upper STREAMS multiplexor driver. This structure is used in conjunction with the `I_LINK`, `I_UNLINK`, `P_LINK`, and `P_UNLINK` ioctl commands. See `streamio`(7I). The `M_DATA` portion of the `M_IOCTL` message contains the `linkblk` structure. Note that the `linkblk` structure is allocated and initialized by the Stream head as a result of one of the above ioctl commands.

Structure Members

```
queue_t   *l_qtop;   /* lowest level write queue of upper stream */
                     /* (set to NULL for persistent links) */
queue_t   *l_qbot;   /* highest level write queue of lower stream */
int       l_index;   /* index for lower stream. */
```

See Also `ioctl`(2), `streamio`(7I)

STREAMS Programming Guide

Name mac_callbacks – MAC callbacks data structure

Synopsis

```
#include <sys/mac_provider.h>
```

Interface Level Solaris architecture specific (Solaris DDI)

Description The `mac_callbacks` data structure is used by MAC device drivers to expose their entry points to the MAC layer. A pointer to an instance of the `mac_callbacks` structure is passed through the *m_callbacks* field of the `mac_register(9S)` structure as part of the registration of a device driver instance through `mac_register(9F)`.

Structure Members

```
uint_t           mc_callbacks;   /* Denotes which callbacks are set */
mac_getstat_t    mc_getstat;     /* Get the value of a statistic */
mac_start_t      mc_start;       /* Start the device */
mac_stop_t       mc_stop;        /* Stop the device */
mac_setpromisc_t mc_setpromisc;  /* Enable or disable promiscuous mode */
mac_multicst_t   mc_multicst;    /* Enable or disable a multicast addr */
mac_unicst_t     mc_unicst;      /* Set the unicast MAC address */
mac_tx_t         mc_tx;          /* Transmit a packet */
mac_ioctl_t      mc_ioctl;       /* Process an unknown ioctl */
mac_getcapab_t   mc_getcapab;    /* Get capability information */
mac_set_prop_t   mc_setprop;     /* Set property value */
mac_get_prop_t   mc_getprop;     /* Get property value */
mac_prop_info_t  mc_propinfo;    /* Get property attributes */
```

Below are descriptions of the members of the `mac_callbacks` structure that are visible to the device driver.

mc_callbacks
: Flags specifying which ones of the optional entry points are implemented by the driver. The following flags are supported:

`MC_IOCTL`
: Set by the driver when the `mc_ioctl()` entry point is present.

`MC_GETCAPAB`
: Set by the driver when the `mc_getcapab()` entry point is present.

`MC_SETPROP`
: Set by the driver when the `mc_setprop()` entry point is present.

`MC_GETPROP`
: Set by the driver when the `mc_getprop()` entry point is present.

`MC_PROPINFO`
: Set by the driver when the `mc_propinfo()` entry point is present.

`MC_PROPERTIES`
: Set by a driver which implements all properties entry points (`mc_setprop()`, `mc_getprop()`, and `mc_propinfo()`). Setting `MC_PROPERTIES` is the equivalent of setting

the three flags MC_SETPROP, MC_GETPROP, and MC_PROPINFO.

mc_getstat
pointer to driver entry point

mc_start
pointer to driver entry point

mc_stop
pointer to driver entry point

mc_setpromisc
pointer to driver entry point

mc_multicst
pointer to driver entry point

mc_unicst
pointer to driver entry point

mc_tx
pointer to driver entry point

mc_ioctl
pointer to driver entry point

mc_getcapab
pointer to driver entry point

mc_setprop
pointer to driver entry point

mc_getprop
pointer to driver entry point

mc_propinfo
pointer to driver entry point

See mac(9E) for more information about MAC driver entry points.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also attributes(5), mac_register(9F), mac_register(9S)

Name mac_capab_lso, lso_basic_tcp_ipv4 – LSO capability data structure

Synopsis `#include <sys/mac_provider.h>`

Interface Level Solaris architecture specific (Solaris DDI)

Description The `mac_capab_lso` and `lso_basic_tcp_ipv4` structures are used by a device driver to describe its LSO capability. The structure is used as the argument to the `mc_getcapab(9E)` driver entry point when querying the `MAC_CAPAB_LSO` capability.

Structure Members The `mac_capab_lso` data structure has the following members:

```
t_uscalar_t             lso_flags;
lso_basic_tcp_ipv4_t    lso_basic_tcp_ipv4;
```

The fields must be set as follows:

lso_flags
Flag indicating the LSO capability supported by the device driver instance. The following flags are currently supported:

`LSO_TX_BASIC_TCP_IPV4` LSO for TCP on IPv4

lso_basic_tcp_ipv4
Parameters for TCP LSO over IPv4

The `lso_basic_tcp_ipv4` data structure is used by the device driver to advertise specific parameters when the `LSO_TX_BASIC_TCP_IPV4` *lso_flag* is set. This data structure has the following elements:

```
t_uscalar_t             lso_max;
```

The *lso_max* field contains the maximum payload size supported by the driver instance.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also `attributes(5)`, `mc_getcapab(9E)`, `mac_lso_get(9F)`, `mac_register(9F)`

Name mac_register – MAC device driver registration data structure

Synopsis

```
#include <sys/mac_provider.h>
#include <sys/mac_ether.h>
```

Interface Level Solaris architecture specific (Solaris DDI)

Description The `mac_register` data structure is passed by device drivers to the MAC layer when registering using `mac_register`(9F).

Structure Members

```
uint_t          m_version;            /* set by framework */
const char      *m_type_ident;
void            *m_driver;
dev_info_t      *m_dip;
uint_t          m_instance;
uint8_t         *m_src_addr;
uint8_t         *m_dst_addr;
mac_callbacks_t *m_callbacks;
uint_t          m_min_sdu;
uint_t          m_max_sdu;
void            *m_pdata;
size_t          m_pdata_size;
mac_priv_prop_t *m_priv_props;
uint32_t          m_margin;
```

The following fields of `mac_register_t` must be set by the device driver before invoking the `mac_register()` entry point:

m_version Set by `mac_alloc`(9F), device drivers should not modify this field.

m_type_ident Must be set to one of the following depending on the type of device being registered.

`MAC_PLUGIN_IDENT_ETHER` Ethernet driver

m_driver Driver handle, opaque to the framework, usually points to a per-driver instance data structure. Passed back as argument to driver's entry points invoked by the framework.

m_dip Pointer to the driver instance `dev_info` structure, see `attach`(9E).

m_instance Used by the driver to specify the instance number to be associated with the MAC being registered. This value should always specified by 0.

m_src_addr Pointer to the primary MAC address value of the MAC instance.

m_dst_addr Pointer to the destination MAC address value of a fixed destination MAC address. This field is optional and should be set to `NULL` for regular device drivers.

m_callbacks Pointer to an instance of the `mac_callbacks`(9S) structure.

m_min_sdu — Minimum Service Data Unit size, the minimum packet size, not including the MAC header, that the device can transmit. This can be zero if the device driver can handle any required padding.

m_max_sdu — Maximum Service Data Unit size, the maximum packet size, not including the MAC header, that can be transmitted by the device. For Ethernet, this number is commonly referred to as the MTU (maximum transmission unit.)

m_priv_props — Array of driver-private property names, terminated by a null pointer.

m_margin — Drivers set this value to the amount of data in bytes that the device can transmit beyond `m_max_sdu`. For example, if an Ethernet device can handle packets whose payload section is no greater than 1522 bytes and `m_max_sdu` is set to 1500 (as is typical for Ethernet), then `m_margin` is set to 22.

See `mac_register(9F)` for more information about the use of these fields.

The driver is responsible for allocating the memory pointed to by the fields *m_priv_props*, *m_src_addr*, and *m_dst_addr*. The driver can free this memory after the call to `mac_register()` returns.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWhea
Interface Stability	Committed

See Also

`attributes(5)`, `attach(9E)`, `mac_register(9F)`, `mac_callbacks(9S)`

Name modldrv – linkage structure for loadable drivers

Synopsis

```
#include <sys/modctl.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `modldrv` structure is used by device drivers to export driver specific information to the kernel.

Structure Members

```
struct mod_ops    *drv_modops;
char              *drv_linkinfo;
struct dev_ops    *drv_dev_ops;
```

`drv_modops` Must always be initialized to the address of `mod_driverops`. This member identifies the module as a loadable driver.

`drv_linkinfo` Can be any string up to `MODMAXNAMELEN` characters (including the terminating NULL character), and is used to describe the module and its version number. This is usually the name of the driver and module version information, but can contain other information as well.

`drv_dev_ops` Pointer to the driver's `dev_ops`(9S) structure.

See Also `add_drv`(1M), `dev_ops`(9S), `modlinkage`(9S)

Writing Device Drivers

Name modlinkage – module linkage structure

Synopsis

```
#include <sys/modctl.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The modlinkage structure is provided by the module writer to the routines that install, remove, and retrieve information from a module. See _init(9E), _fini(9E), and _info(9E).

Structure Members

```
int     ml_rev
void    *ml_linkage[4];
```

ml_rev — Is the revision of the loadable modules system. This must have the value MODREV_1.

ml_linkage — Is a null-terminated array of pointers to linkage structures. Driver modules have only one linkage structure.

See Also add_drv(1M), _fini(9E), _info(9E), _init(9E), modldrv(9S), modlstrmod(9S)

Writing Device Drivers

Name modlstrmod – linkage structure for loadable STREAMS modules

Synopsis

```
#include <sys/modctl.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `modlstrmod` structure is used by STREAMS modules to export module specific information to the kernel.

Structure Members

```
struct mod_ops     *strmod_modops;
char               *strmod_linkinfo;
struct fmodsw      *strmod_fmodsw;
```

`strmod_modops` Must always be initialized to the address of `mod_strmodops`. This identifies the module as a loadable STREAMS module.

`strmod_linkinfo` Can be any string up to `MODMAXNAMELEN`, and is used to describe the module. This string is usually the name of the module, but can contain other information (such as a version number).

`strmod_fmodsw` Is a pointer to a template of a class entry within the module that is copied to the kernel's class table when the module is loaded.

See Also `modload`(1M)

Writing Device Drivers

Name module_info – STREAMS driver identification and limit value structure

Synopsis `#include <sys/stream.h>`

Interface Level Architecture independent level 1 (DDI/DKI).

Description When a module or driver is declared, several identification and limit values can be set. These values are stored in the `module_info` structure.

The `module_info` structure is intended to be read-only. However, the flow control limits (`mi_hiwat` and `mi_lowat`) and the packet size limits (`mi_minpsz` and `mi_maxpsz`) are copied to the `QUEUE` structure, where they can be modified.

For a driver, `mi_idname` must match the name of the driver binary file. For a module, `mi_idname` must match the fname field of the `fmodsw` structure. See `fmodsw`(9S) for details.

Structure Members

```
ushort_t      mi_idnum;      /* module ID number */
char          *mi_idname;    /* module name */
ssize_t       mi_minpsz;     /* minimum packet size */
ssize_t       mi_maxpsz;     /* maximum packet size */
size_t        mi_hiwat;      /* high water mark */
size_t        mi_lowat;      /* low water mark */
```

The constant `FMNAMESZ`, limiting the length of a module's name, is set to eight in this release.

See Also `fmodsw`(9S), `queue`(9S)

STREAMS Programming Guide

Name msgb, mblk – STREAMS message block structure

Synopsis `#include <sys/stream.h>`

Interface Level Architecture independent level 1 (DDI/DKI)

Description A STREAMS message is made up of one or more message blocks, referenced by a pointer to a `msgb` structure. The `b_next` and `b_prev` pointers are used to link messages together on a `QUEUE`. The `b_cont` pointer links message blocks together when a message consists of more than one block.

Each `msgb` structure also includes a pointer to a `datab(9S)` structure, the data block (which contains pointers to the actual data of the message), and the type of the message.

Structure Members

```
struct msgb     *b_next;    /* next message on queue */
struct msgb     *b_prev;    /* previous message on queue */
struct msgb     *b_cont;    /* next message block */
unsigned char   *b_rptr;    /* 1st unread data byte of buffer */
unsigned char   *b_wptr;    /* 1st unwritten data byte of buffer */
struct datab    *b_datap;   /* pointer to data block */
unsigned char   b_band;     /* message priority  */
unsigned short  b_flag;     /* used by stream head  */
```

Valid flags are as follows:

`MSGMARK` Last byte of message is marked.

`MSGDELIM` Message is delimited.

The `msgb` structure is defined as type `mblk_t`.

See Also `datab(9S)`

Writing Device Drivers

STREAMS Programming Guide

Name net_inject_t – structure for describing how to transmit a packet

Synopsis

```
#include <sys/neti.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The `net_inject_t` data structure passes information in to `net_inject` about how to transmit a packet. Transmit includes sending the packet up into the system as well as out of it.

Structure Members

```
mblk_t                  *ni_packet;   /* start of the packet */
 struct sockaddr_storage ni_addr;      /* address of next hop */
 phy_if_t                ni_physical;  /* network interface to use */
```

`ni_packet` Pointer to the first the `mblk_t` data structure that makes up this packet.

`ni_addr` This field is only required to be initialized if NI_DIRECT_OUT is being used to transmit the packet. The `sockaddr_storage` field must be set to indicate whether the destination address contained in the structure is IPv4 (cast *ni_addr* to struct *sockaddr_in*) or IPv6 (cast *ni_addr* to struct *sockaddr_in6*).

`ni_physical` The physical interface where the packet will be injected.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `net_inject(9F)`, `netinfo(9F)`, `attributes(5)`

Name net_instance_t – packet event structure passed through to hooks

Synopsis

```
#include <sys/neti.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The net_instance_t data structure defines a collection of instances to be called when relevant events happen within IP. The value returned by the nin_create() function is stored internally and passed back to both the nin_destroy() and nin_shutdown() functions as the second argument. The netid_t passed through to each function can be used to uniquely identify each instance of IP.

Structure Members

```
char    *nin_name;
 void    *(*nin_create)(const netid_t);
 void    (*nin_destroy)(const netid_t, void *);
 void    (*nin_shutdown)(const netid_t, void *);
```

nin_name — Name of the owner of the instance.

nin_create — Function to be called when a new instance of IP is created.

nin_destroy — Function to be called when an instance of IP is being destroyed.

nin_shutdown — Function to be called when an instance of IP is being shutdown. nin_shutdown() is called before nin_destroy() is called.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also netinfo(9F), attributes(5)

Name

no-involuntary-power-cycles – device property to prevent involuntary power cycles

Description

A device that might be damaged by power cycles should export the boolean (zero length) property `no-involuntary-power-cycles` to notify the system that all power cycles for the device must be under the control of the device driver.

The presence of this property prevents power from being removed from a device or any ancestor of the device while the device driver is detached, unless the device was voluntarily powered off as a result of the device driver calling `pm_lower_power`(9F).

The presence of `no-involuntary-power-cycles` also forces attachment of the device driver during a CPR suspend operation and prevents the suspend from taking place, unless the device driver returns `DDI_SUCCESS` when its `detach`(9E) entry point is called with `DDI_SUSPEND`.

The presence of `no-involuntary-power-cycles` does not prevent the system from being powered off due to a `halt`(1M) or `uadmin`(1M) invocation, except for CPR suspend.

This property can be exported by a device that is not power manageable, in which case power is not removed from the device or from any of its ancestors, even when the driver for the device and the drivers for its ancestors are detached.

Examples

EXAMPLE 1 Use of Property in Driver's Configuration File

The following is an example of a `no-involuntary-power-cycles` entry in a driver's `.conf` file:

```
no-involuntary-power-cycles=1;
     ...
```

EXAMPLE 2 Use of Property in `attach()` Function

The following is an example of how the preceding `.conf` file entry would be implemented in the `attach`(9E) function of a driver:

```
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
  {
   ...
       if (ddi_prop_create(DDI_DEV_T_NONE, dip, DDI_PROP_CANSLEEP,
           "no-involuntary-power-cycles", NULL, 0) != DDI_PROP_SUCCESS)
            goto failed;
   ...
   }
```

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

See Also attributes(5), pm(7D), attach(9E), detach(9E), ddi_prop_create(9F)

Writing Device Drivers

Name pm – Power Management properties

Description The pm-hardware-state property can be used to influence the behavior of the Power Management framework. Its syntax and interpretation is described below.

Note that this property is only interpreted by the system immediately after the device has successfully attached. Changes in the property made by the driver after the driver has attached will not be recognized.

pm-hardware-state is a string-valued property. The existence of the pm-hardware-state property indicates that a device needs special handling by the Power Management framework with regard to its hardware state.

If the value of this property is needs-suspend-resume, the device has a hardware state that cannot be deduced by the framework. The framework definition of a device with hardware state is one with a reg property. Some drivers, such as SCSI disk and tape drivers, have no reg property but manage devices with "remote" hardware. Such a device must have a pm-hardware-state property with a value of needs-suspend-resume for the system to identify it as needing a call to its detach(9E) entry point with command DDI_SUSPEND when system is suspended, and a call to attach(9E) with command DDI_RESUME when system is resumed. For devices using original Power Management interfaces (which are now obsolete) detach(9E) is also called with DDI_PM_SUSPEND before power is removed from the device, and attach(9E) is called with DDI_PM_RESUME after power is restored.

A value of no-suspend-resume indicates that, in spite of the existence of a reg property, a device has no hardware state that needs saving and restoring. A device exporting this property will not have its detach() entry point called with command DDI_SUSPEND when system is suspended, nor will its attach() entry point be called with command DDI_RESUME when system is resumed. For devices using the original (and now obsolete) Power Management interfaces, detach(9E) will not be called with DDI_PM_SUSPEND command before power is removed from the device, nor attach(9E) will be called with DDI_PM_RESUME command after power is restored to the device.

A value of parental-suspend-resume indicates that the device does not implement the detach(9E) DDI_SUSPEND semantics, nor the attach() DDI_RESUME semantics, but that a call should be made up the device tree by the framework to effect the saving and/or restoring of hardware state for this device. For devices using original Power Management interfaces (which are now obsolete), it also indicates that the device does not implement the detach(9E) DDI_PM_SUSPEND semantics, nor the attach(9E) DDI_PM_RESUME semantics, but that a call should be made up the device tree by the framework to effect the saving and/or restoring the hardware state for this device.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

See Also power.conf(4), pm(7D), attach(9E), detach(9E), pm_busy_component(9F), pm_idle_component(9F), pm-components(9P)

Writing Device Drivers

Name pm-components – Power Management device property

Description A device is power manageable if the power consumption of the device can be reduced when it is idle. In general, a power manageable device consists of a number of power manageable hardware units called components. Each component is separately controllable and has its own set of power parameters.

An example of a one-component power manageable device is a disk whose spindle motor can be stopped to save power when the disk is idle. An example of a two-component power manageable device is a frame buffer card with a connected monitor. The frame buffer electronics (with power that can be reduced when not in use) comprises the first component. The second component is the monitor, which can enter in a lower power mode when not in use. The combination of frame buffer electronics and monitor is considered as one device by the system.

In the Power Management framework, all components are considered equal and completely independent of each other. If this is not true for a particular device, the device driver must ensure that undesirable state combinations do not occur. Each component is created in the idle state.

The `pm-components` property describes the Power Management model of a device driver to the Power Management framework. It lists each power manageable component by name and lists the power level supported by each component by numerical value and name. Its syntax and interpretation is described below.

This property is only interpreted by the system immediately after the device has successfully attached, or upon the first call into Power Management framework, whichever comes first. Changes in the property made by the driver after the property has been interpreted will not be recognized.

`pm-components` is a string array property. The existence of the `pm-components` property indicates that a device implements power manageable components and describes the Power Management model implemented by the device driver. The existence of `pm-components` also indicates to the framework that device is ready for Power Management if automatic device Power Management is enabled. See `power.conf(4)`.

The `pm-component` property syntax is:

```
pm-components="NAME=component name", "numeric power level=power level name",
  "numeric power level=power level name"
  [, "numeric power level=power level name" ...]
  [, "NAME=component name", "numeric power level=power level name",
  "numeric power level=power level name"
  [, "numeric power level=power level name"...]...];
```

The start of each new component is represented by a string consisting of `NAME=` followed by the name of the component. This should be a short name that a user would recognize, such as "Monitor" or "Spindle Motor." The succeeding elements in the string array must be strings consisting of the numeric value (can be decimal or 0x <hexadecimal number>) of a power level the component supports, followed by an equal sign followed by a short descriptive name for that power level. Again, the names should be descriptive, such as "On," "Off," "Suspend," "Standby," etc. The next component continues the array in the same manner, with a string that starts out NAME=, specifying the beginning of a new component (and its name), followed by specifications of the power levels the component supports.

The components must be listed in increasing order according to the component number as interpreted by the driver's power(9E) routine. (Components are numbered sequentially from 0). The power levels must be listed in increasing order of power consumption. Each component must support at least two power levels, or there is no possiblity of power level transitions. If a power level value of 0 is used, it must be the first one listed for that component. A power level value of 0 has a special meaning (off) to the Power Management framework.

Examples

An example of a `pm-components` entry from the `.conf` file of a driver which implements a single power managed component consisting of a disk spindle motor is shown below. This is component 0 and it supports 2 power level, which represent spindle stopped or full speed.

```
pm-components="NAME=Spindle Motor", "0=Stopped", "1=Full Speed";
...
```

Below is an example of how the above entry would be implemented in the attach(9E) function of the driver.

```
static char *pmcomps[] = {
  "NAME=Spindle Motor",
     "0=Stopped",
     "1=Full Speed"
};

...

xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
...
     if (ddi_prop_update_string_array(DDI_DEV_T_NONE, dip, "pm-components",
        &pmcomp[0], sizeof (pmcomps) / sizeof (char *)) != DDI_PROP_SUCCESS)
        goto failed;
}
```

Below is an example for a frame buffer which implements two components. Component 0 is the frame buffer electronics which supports four different power levels. Component 1 represents the state of Power Management of the attached monitor.

```
pm-components="NAME=Frame Buffer", "0=Off"
     "1=Suspend", "2=Standby", "3=On",
        "NAME=Monitor", "0=Off", "1=Suspend", "2=Standby,"
        "3=On;
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

See Also power.conf(4), pm(7D), attach(9E), detach(9E), ddi_prop_update_string_array(9F) pm_busy_component(9F), pm_idle_component(9F)

Writing Device Drivers

Name qband – STREAMS queue flow control information structure

Synopsis

```
#include <sys/stream.h>
```

Interface Level Architecture independent level 1 (DDI/DKI)

Description The qband structure contains flow control information for each priority band in a queue.

The qband structure is defined as type qband_t.

Structure Members

```
struct          qband*qb_next;       /* next band's info */
size_t          qb_count             /* number of bytes in band */
struct msgb     *qb_first;           /* start of band's data */
struct msgb     *qb_last;            /* end of band's data */
size_t          qb_hiwat;            /* band's high water mark */
size_t          qb_lowat;            /* band's low water mark */
uint_t          qb_flag;             /* see below */
```

Valid flags are as follows:

QB_FULL Band is considered full.

QB_WANTW Someone wants to write to band.

See Also strqget(9F), strqset(9F), msgb(9S), queue(9S)

STREAMS Programming Guide

Notes All access to this structure should be through strqget(9F) and strqset(9F). It is logically part of the queue(9S) and its layout and partitioning with respect to that structure might change in future releases. If portability is a concern, do not declare or store instances of or references to this structure.

Name qinit – STREAMS queue processing procedures structure

Synopsis

```
#include <sys/stream.h>
```

Interface Level Architecture independent level 1 (DDI/DKI)

Description The `qinit` structure contains pointers to processing procedures for a `QUEUE`. The `streamtab` structure for the module or driver contains pointers to one `queue(9S)` structure for both upstream and downstream processing.

Structure Members

```
int                   (*qi_putp)();      /* put procedure */
int                   (*qi_srvp)();      /* service procedure */
int                   (*qi_qopen)();     /* open procedure */
int                   (*qi_qclose)();    /* close procedure */
int                   (*qi_qadmin)();    /* unused */
struct module_info  *qi_minfo;         /* module parameters */
struct module_stat  *qi_mstat;         /* module statistics */
```

See Also `queue(9S)`, `streamtab(9S)`

Writing Device Drivers

STREAMS Programming Guide

Notes This release includes no support for module statistics.

Name queclass – a STREAMS macro that returns the queue message class definitions for a given message block

Synopsis

```
#include <sys/stream.h>

queclass(mblk_t *bp);
```

Interface Level Solaris DDI specific (Solaris DDI)

Description `queclass` returns the queue message class definition for a given data block pointed to by the message block *bp* passed in.

The message can be either `QNORM`, a normal priority message, or `QPCTL`, a high priority message.

See Also *STREAMS Programming Guide*

Name queue – STREAMS queue structure

Synopsis `#include <sys/stream.h>`

Interface Level Architecture independent level 1 (DDI/DKI)

Description A STREAMS driver or module consists of two queue structures: *read* for upstream processing and *write* for downstream processing. The queue structure is the major building block of a stream.

queue Structure Members The queue structure is defined as type queue_t. The structure can be accessed at any time from inside a STREAMS entry point associated with that queue.

```
struct    qinit   *q_qinfo;     /* queue processing procedure */
struct    msgb    *q_first;     /* first message in queue */
struct    msgb    *q_last;      /* last message in queue */
struct    queue   *q_next;      /* next queue in stream */
void              *q_ptr;       /* module-specific data */
size_t             q_count;     /* number of bytes on queue */
uint_t             q_flag;      /* queue state */
ssize_t            q_minpsz;    /* smallest packet OK on queue */
ssize_t            q_maxpsz;    /* largest packet OK on queue */
size_t             q_hiwat;     /* queue high water mark */
size_t             q_lowat;     /* queue low water mark */
```

Contstraints and restrictions on the use of q_flag and queue_t fields and the q_next values are detailed in the following sections.

q_flag Field The q_flag field must be used only to check the following flag values.

QFULL Queue is full.

QREADR Queue is used for upstream (read-side) processing.

QUSE Queue has been allocated.

QENAB Queue has been enabled for service by qenable(9F).

QNOENB Queue will not be scheduled for service by putq(9F).

QWANTR Upstream processing element wants to read from queue.

QWANTW Downstream processing element wants to write to queue.

queue_t Fields Aside from q_ptr and q_qinfo, a module or driver must never assume that a queue_t field value will remain unchanged across calls to STREAMS entry points. In addition, many fields can change values inside a STREAMS entry point, especially if the STREAMS module or driver has perimeters that allow parallelism. See mt-streams(9F). Fields that are not documented below are private to the STREAMS framework and must not be accessed.

- The values of the q_hiwat, q_lowat, q_minpsz, and q_maxpsz fields can be changed at the discretion of the module or driver. As such, the stability of their values depends on the perimeter configuration associated with any routines that modify them.
- The values of the q_first, q_last, and q_count fields can change whenever putq(9F), putbq(9F), getq(9F), insq(9F), or rmvq(9F) is used on the queue. As such, the stability of their values depends on the perimeter configuration associated with any routines that call those STREAMS functions.
- The q_flag field can change at any time.
- The q_next field will not change while inside a given STREAMS entry point. Additional restrictions on the use of the q_next value are described in the next section.

A STREAMS module or driver can assign any value to q_ptr. Typically q_ptr is used to point to module-specific per-queue state, allocated in open(9E) and freed in close(9E). The value or contents of q_ptr is never inspected by the STREAMS framework.

The initial values for q_minpsz, q_maxpsz, q_hiwat, and q_lowat are set using the module_info(9S) structure when mod_install(9F) is called. A STREAMS module or driver can subsequently change the values of those fields as necessary. The remaining visible fields, q_qinfo, q_first, q_last, q_next, q_count, and q_flag, must never be modified by a module or driver.

The Solaris DDI requires that STREAMS modules and drivers obey the rules described on this page. Those that do not follow the rules can cause data corruption or system instability, and might change in behavior across patches or upgrades.

q_next Restrictions

There are additional restrictions associated with the use of the q_next value. In particular, a STREAMS module or driver:

- Must not access the data structure pointed to by q_next.
- Must not rely on the value of q_next before calling qprocson(9F) or after calling qprocsoff(9F).
- Must not pass the value into any STREAMS framework function other than put(9F), canput(9F), bcanput(9F), putctl(9F), putctl1(9F). However, in all cases the "next" version of these functions, such as putnext(9F), should be preferred.
- Must not use the value to compare against queue pointers from other streams. However, checking q_next for NULL can be used to distinguish a module from a driver in code shared by both.

See Also

close(9E), open(9E), bcanput(9F), canput(9F), getq(9F), insq(9F), mod_install(9F), put(9F), putbq(9F), putctl(9F), putctl1(9F), putnext(9F), putq(9F), qprocsoff(9F), qprocson(9F), rmvq(9F), strqget(9F), strqset(9F), module_info(9S), msgb(9S), qinit(9S), streamtab(9S)

Writing Device Drivers

STREAMS Programming Guide

Name removable-media – removable media device property

Description A device that supports removable media—such as CDROM, JAZZ, and ZIP drives—and that supports power management and expects automatic mounting of the device via the volume manager should export the boolean (zero length) property `removable-media`. This property enables the system to make the power state of the device dependent on the power state of the frame buffer and monitor. See the `power.conf`(4) discussion of the `device-dependency-property` entry for more information.

Devices that behave like removable devices (such as PC ATA cards, where the controller and media both are removed at the same time) should also export this property.

Examples **EXAMPLE 1** `removable-media` Entry

An example of a `removable-media` entry from the `.conf` file of a driver is shown below.

```
# This entry keeps removable media from being powered down unless
# the console framebuffer and monitor are powered down
#
removable-media=1;
```

EXAMPLE 2 Implementation in `attach()`

Below is an example of how the entry above would be implemented in the `attach`(9E) function of the driver.

```
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
     {
     ...
          if (ddi_prop_create(DDI_DEV_T_NONE, dip, DDI_PROP_CANSLEEP,
              "removable-media", NULL, 0)) != DDI_PROP_SUCCESS)
               goto failed;
     ...
     }
```

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving

See Also `power.conf`(4), `pm`(7D), `attach`(9E), `detach`(9E), `ddi_prop_create`(9F)

Writing Device Drivers

Name scsi_address – SCSI address structure

Synopsis

```
#include <sys/scsi/scsi.h>
```

Interface Level Solaris architecture specific (Solaris DDI)

Description A `scsi_address` structure defines the addressing components for a SCSI target device. The address of the target device is separated into two components: target number and logical unit number. The two addressing components are used to uniquely identify any type of SCSI device; however, most devices can be addressed with the target component of the address.

In the case where only the target component is used to address the device, the logical unit should be set to `0`. If the SCSI target device supports logical units, then the HBA must interpret the logical units field of the data structure.

The `pkt_address` member of a `scsi_pkt(9S)` is initialized by `scsi_init_pkt(9F)`.

Structure Members

```
scsi_hba_tran_t   *a_hba_tran;  /* Transport vectors for the SCSI bus */
ushort_t          a_target;     /* SCSI target id */
uchar_t           a_lun;        /* SCSI logical unit */
```

`a_hba_tran` is a pointer to the controlling HBA's transport vector structure. The SCSA interface uses this field to pass any transport requests from the SCSI target device drivers to the HBA driver.

`a_target` is the target component of the SCSI address.

`a_lun` is the logical unit component of the SCSI address. The logical unit is used to further distinguish a SCSI target device that supports multiple logical units from one that does not. The `makecom(9F)` family of functions use the `a_lun` field to set the logical unit field in the SCSI CDB, for compatibility with SCSI-1.

See Also `makecom(9F)`, `scsi_init_pkt(9F)`, `scsi_hba_tran(9S)`, `scsi_pkt(9S)`

Writing Device Drivers

Name scsi_arq_status – SCSI auto request sense structure

Synopsis `#include <sys/scsi/scsi.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description When auto request sense has been enabled using `scsi_ifsetcap`(9F) and the "auto-rqsense" capability, the target driver must allocate a status area in the SCSI packet structure for the auto request sense structure (see `scsi_pkt`(9S)). In the event of a `check` *condition*, the transport layer automatically executes a request sense command. This check ensures that the request sense information does not get lost. The auto request sense structure supplies the SCSI status of the original command, the transport information pertaining to the request sense command, and the request sense data.

Structure Members

```
struct scsi_status       sts_status;          /* SCSI status */
struct scsi_status       sts_rqpkt_status;    /* SCSI status of
                                                 request sense cmd */
uchar_t                  sts_rqpkt_reason;    /* reason completion */
uchar_t                  sts_rqpkt_resid;     /* residue */
uint_t                   sts_rqpkt_state;     /* state of command */
uint_t                   sts_rqpkt_statistics;/* statistics */
struct scsi_extended_sense sts_sensedata;      /* actual sense data */
```

`sts_status` is the SCSI status of the original command. If the status indicates a `check` *condition*, the transport layer might have performed an auto request sense command.

`sts_rqpkt_status` is the SCSI status of the request sense command. `sts_rqpkt_reason` is the completion reason of the request sense command. If the reason is not `CMD_CMPLT`, then the request sense command did not complete normally.

`sts_rqpkt_resid` is the residual count of the data transfer and indicates the number of data bytes that have not been transferred. The auto request sense command requests `SENSE_LENGTH` bytes.

`sts_rqpkt_state` has bit positions representing the five most important statuses that a SCSI command can go obtain.

`sts_rqpkt_statistics` maintains transport-related statistics of the request sense command.

`sts_sensedata` contains the actual sense data if the request sense command completed normally.

See Also `scsi_ifgetcap`(9F), `scsi_init_pkt`(9F), `scsi_extended_sense`(9S), `scsi_pkt`(9S)

Writing Device Drivers

Name scsi_asc_key_strings – SCSI ASC ASCQ to message structure

Synopsis
```
#include <sys/scsi/scsi.h>
```

Interface Level Solaris DDI specific (Solaris DDI).

Description The `scsi_asc_key_strings` structure stores the ASC and ASCQ codes and a pointer to the related ASCII string.

Structure Members
```
ushort_t asc;           /* ASC code */
ushort_t ascq;          /* ASCQ code */
char    *message;       /* ASCII message string */

asc               Contains the ASC key code.

ascq              Contains the ASCQ code.

message           Points to the NULL terminated ASCII string
                  describing the asc and ascq condition
```

See Also `scsi_vu_errmsg`(9F)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

Name scsi_device – SCSI device structure

Synopsis `#include <sys/scsi/scsi.h>`

Interface Level Solaris DDI specific (Solaris DDI).

Description The `scsi_device` structure stores common information about each SCSI logical unit, including pointers to areas that contain both generic and device specific information. There is one `scsi_device` structure for each logical unit attached to the system. The host adapter driver initializes part of this structure prior to `probe`(9E) and destroys this structure after a probe failure or successful `detach`(9E).

Structure Members

```
struct scsi_address        sd_address; /* Routing info. */
dev_info_t                 *sd_dev;    /* Cross-ref. to */
                                       /* dev_info_t */
kmutex_t                   sd_mutex;   /* Mutex for this dev. */
struct scsi_inquiry        *sd_inq;    /* scsi_inquiry data struc. */
struct scsi_extended_sense *sd_sense;  /* Optional request */
                                       /* sense buffer ptr */
caddr_t                    sd_private; /* Target drivers
                                          private data */
```

`sd_address` contains the routing information that the target driver normally copies into a `scsi_pkt`(9S) structure using the collection of `makecom`(9F) functions. The SCSA library routines use this information to determine which host adapter, SCSI bus, and target/logical unit number (lun) a command is intended for. This structure is initialized by the host adapter driver.

`sd_dev` is a pointer to the corresponding `dev_info` structure. This pointer is initialized by the host adapter driver.

`sd_mutex` is a mutual exclusion lock for this device. It is used to serialize access to a device. The host adapter driver initializes this mutex. See `mutex`(9F).

`sd_inq` is initially `NULL` (zero). After executing `scsi_probe`(9F), this field contains the inquiry data associated with the particular device.

`sd_sense` is initially `NULL` (zero). If the target driver wants to use this field for storing REQUEST SENSE data, it should allocate an `scsi_extended_sense`(9S) buffer and set this field to the address of this buffer.

`sd_private` is reserved for the use of target drivers and should generally be used to point to target specific data structures.

See Also `detach`(9E), `probe`(9E), `makecom`(9F), `mutex`(9F), `scsi_probe`(9F), `scsi_extended_sense`(9S), `scsi_pkt`(9S)

Writing Device Drivers

Name scsi_extended_sense – SCSI extended sense structure

Synopsis `#include <sys/scsi/scsi.h>`

Interface Level Solaris DDI specific (Solaris DDI).

Description The `scsi_extended_sense` structure for error codes `0x70` (current errors) and `0x71` (deferred errors) is returned on a successful `REQUEST SENSE` command. SCSI-2 compliant targets are required to return at least the first 18 bytes of this structure. This structure is part of `scsi_device(9S)` structure.

Structure Members

```
uchar_t  es_valid   :1;     /* Sense data is valid */
uchar_t  es_class   :3;     /* Error Class- fixed at 0x7 */
uchar_t  es_code    :4;     /* Vendor Unique error code */
uchar_t  es_segnum;         /* Segment number: for COPY cmd only */
uchar_t  es_filmk   :1;     /* File Mark Detected */
uchar_t  es_eom     :1;     /* End of Media */
uchar_t  es_ili     :1;     /* Incorrect Length Indicator */
uchar_t  es_key     :4;     /* Sense key */
uchar_t  es_info_1;         /* Information byte 1 */
uchar_t  es_info_2;         /* Information byte 2 */
uchar_t  es_info_3;         /* Information byte 3 */
uchar_t  es_info_4;         /* Information byte 4 */
uchar_t  es_add_len;        /* Number of additional bytes */
uchar_t  es_cmd_info[4];    /* Command specific information */
uchar_t  es_add_code;       /* Additional Sense Code */
uchar_t  es_qual_code;      /* Additional Sense Code Qualifier */
uchar_t  es_fru_code;       /* Field Replaceable Unit Code */
uchar_t  es_skey_specific[3]; /* Sense Key Specific information */
```

`es_valid`, if set, indicates that the information field contains valid information.

`es_class` should be `0x7`.

`es_code` is either `0x0` or `0x1`.

`es_segnum` contains the number of the current segment descriptor if the `REQUEST SENSE` command is in response to a `COPY`, `COMPARE`, and `COPY AND VERIFY` command.

`es_filmk`, if set, indicates that the current command had read a file mark or set mark (sequential access devices only).

`es_eom`, if set, indicates that an end-of-medium condition exists (sequential access and printer devices only).

`es_ili`, if set, indicates that the requested logical block length did not match the logical block length of the data on the medium.

`es_key` indicates generic information describing an error or exception condition. The following sense keys are defined:

KEY_NO_SENSE	Indicates that there is no specific sense key information to be reported.
KEY_RECOVERABLE_ERROR	Indicates that the last command completed successfully with some recovery action performed by the target.
KEY_NOT_READY	Indicates that the logical unit addressed cannot be accessed.
KEY_MEDIUM_ERROR	Indicates that the command terminated with a non-recovered error condition that was probably caused by a flaw on the medium or an error in the recorded data.
KEY_HARDWARE_ERROR	Indicates that the target detected a non-recoverable hardware failure while performing the command or during a self test.
KEY_ILLEGAL_REQUEST	Indicates that there was an illegal parameter in the CDB or in the additional parameters supplied as data for some commands.
KEY_UNIT_ATTENTION	Indicates that the removable medium might have been changed or the target has been reset.
KEY_WRITE_PROTECT/KEY_DATA_PROTECT	Indicates that a command that reads or writes the medium was attempted on a block that is protected from this operation.
KEY_BLANK_CHECK	Indicates that a write-once device or a sequential access device encountered blank medium or format-defined end-of-data indication while reading or a write-once device encountered a non-blank medium while writing.
KEY_VENDOR_UNIQUE	This sense key is available for reporting vendor-specific conditions.
KEY_COPY_ABORTED	Indicates that a COPY, COMPARE, and COPY AND VERIFY command was aborted.
KEY_ABORTED_COMMAND	Indicates that the target aborted the command.
KEY_EQUAL	Indicates that a SEARCH DATA command has satisfied an equal comparison.

KEY_VOLUME_OVERFLOW	Indicates that a buffered peripheral device has reached the end-of-partition and data might remain in the buffer that has not been written to the medium.
KEY_MISCOMPARE	Indicates that the source data did not match the data read from the medium.
KEY_RESERVE	Indicates that the target is currently reserved by a different initiator.

es_info_{1,2,3,4} is device-type or command specific.

es_add_len indicates the number of additional sense bytes to follow.

es_cmd_info contains information that depends on the command that was executed.

es_add_code (ASC) indicates further information related to the error or exception condition reported in the sense key field.

es_qual_code (ASCQ) indicates detailed information related to the additional sense code.

es_fru_code (FRU) indicates a device-specific mechanism to unit that has failed.

es_skey_specific is defined when the value of the sense-key specific valid bit (bit 7) is 1. This field is reserved for sense keys not defined above.

See Also

scsi_device(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

Writing Device Drivers

Name scsi_hba_tran – SCSI Host Bus Adapter (HBA) driver transport vector structure

Synopsis

```
#include <sys/scsi/scsi.h>
```

Interface Level Solaris architecture specific (Solaris DDI).

Description A `scsi_hba_tran_t` structure defines vectors that an HBA driver exports to SCSA interfaces so that HBA specific functions can be executed.

Structure Members

```
dev_info_t          *tran_hba_dip;          /* HBAs dev_info
                                                pointer */
void                *tran_hba_private;      /* HBA softstate */
void                *tran_tgt_private;      /* HBA target private
                                                pointer */
struct scsi_device  *tran_sd;               /* scsi_device */
int                 (*tran_tgt_init)( );    /* Transport target */
                                                Initialization */
int                 (*tran_tgt_probe)( );   /* Transport target
                                                probe */
void                (*tran_tgt_free)( );    /* Transport target
                                                free */
int                 (*tran_start)( );       /* Transport start */
int                 (*tran_reset)( );       /* Transport reset */
int                 (*tran_abort)( );       /* Transport abort */
int                 (*tran_getcap)( );      /* Capability
                                                retrieval */
int                 (*tran_setcap)( );      /* Capability
                                                establishment */
struct scsi_pkt     *(*tran_init_pkt)( );   /* Packet and DMA
                                                allocation */
void                (*tran_destroy_pkt)( ); /* Packet and DMA */
                                                deallocation */
void                (*tran_dmafree)( );     /* DMA deallocation */
void                (*tran_sync_pkt)( );    /* Sync DMA */
void                (*tran_reset_notify)( );/* Bus reset
                                                notification */
int                 (*tran_bus_reset)( );   /* Reset bus only */
int                 (*tran_quiesce)( );     /* Quiesce a bus */
int                 (*tran_unquiesce)( );   /* Unquiesce a bus */
int                 (*tran_setup_pkt)( );   /* Initialization
                                                for pkt */
int                 (*tran_teardown_pkt)( );/* Deallocation */
int                 (*tran_pkt_constructor) ( );
                                              /* Constructor */
int                 (*tran_pkt_destructor) ( );
                                             /* Destructor */
int                 tran_hba_len;             /* # bytes for
                                                 pkt_ha_private */
```

```
int                 tran_interconnect_type;  /* transport
                                                interconnect */
```

`tran_hba_dip` — `dev_info` pointer to the HBA that supplies the `scsi_hba_tran` structure.

`tran_hba_private` — Private pointer that the HBA driver can use to refer to the device's soft state structure.

`tran_tgt_private` — Private pointer that the HBA can use to refer to per-target specific data. This field can only be used when the `SCSI_HBA_TRAN_CLONE` flag is specified in `scsi_hba_attach`(9F). In this case, the HBA driver must initialize this field in its `tran_tgt_init`(9E) entry point.

`tran_sd` — Pointer to `scsi_device`(9S) structure if cloning; otherwise `NULL`.

`tran_tgt_init` — Function entry that allows per-target HBA initialization, if necessary.

`tran_tgt_probe` — Function entry that allows per-target `scsi_probe`(9F) customization, if necessary.

`tran_tgt_free` — Function entry that allows per-target HBA deallocation, if necessary.

`tran_start` — Function entry that starts a SCSI command execution on the HBA hardware.

`tran_reset` — Function entry that resets a SCSI bus or target device.

`tran_abort` — Function entry that aborts one SCSI command, or all pending SCSI commands.

`tran_getcap` — Function entry that retrieves a SCSI capability.

`tran_setcap` — Function entry that sets a SCSI capability.

`tran_init_pkt` — Function entry that allocates a `scsi_pkt` structure.

`tran_destroy_pkt` — Function entry that frees a `scsi_pkt` structure allocated by `tran_init_pkt`.

`tran_dmafree` — Function entry that frees DMA resources that were previously allocated by `tran_init_pkt`. Not called for HBA drivers that provide a `tran_setup_pkt` entry point.

`tran_sync_pkt` — Synchronizes data in *pkt* after a data transfer has been completed. Not called for HBA drivers that provide a `tran_setup_pkt` entry point.

tran_reset_notify	Function entry that allows a target to register a bus reset notification request with the HBA driver.
tran_bus_reset	Function entry that resets the SCSI bus without resetting targets.
tran_quiesce	Function entry that waits for all outstanding commands to complete and blocks (or queues) any I/O requests issued.
tran_unquiesce	Function entry that allows I/O activities to resume on the SCSI bus.
tran_setup_pkt	Optional entry point that initializes a scsi_pkt structure.
tran_teardown_pkt	Entry point that releases resources allocated by tran_setup_pkt.
tran_pkt_constructor	Additional optional entry point that performs the actions of a constructor..
tran_pkt_destructor	Additional optional entry point that performs the actions of a destructor..
tran_hba_len	Size of pkt_ha_private..
tran_interconnect_type	Integer value that denotes the interconnect type of the transport as defined in the services.h header file.

See Also tran_abort(9E), tran_bus_reset(9E), tran_destroy_pkt(9E), tran_dmafree(9E), tran_getcap(9E), tran_init_pkt(9E), tran_quiesce(9E), tran_reset(9E), tran_reset_notify(9E), tran_setcap(9E), tran_start(9E), tran_sync_pkt(9E), tran_tgt_free(9E), tran_tgt_init(9E), tran_tgt_probe(9E), tran_unquiesce(9E), ddi_dma_sync(9F), scsi_hba_attach(9F), scsi_hba_pkt_alloc(9F), scsi_hba_pkt_free(9F), scsi_probe(9F), scsi_device(9S), scsi_pkt(9S)

Writing Device Drivers

Name scsi_inquiry – SCSI inquiry structure

Synopsis `#include <sys/scsi/scsi.h>`

Interface Level Solaris DDI specific (Solaris DDI).

Description The `scsi_inquiry` structure contains 36 required bytes, followed by a variable number of vendor-specific parameters. Bytes 59 through 95, if returned, are reserved for future standardization. This structure is part of `scsi_device(9S)` structure and typically filled in by `scsi_probe(9F)`.

Structure Members

```
Lines that start with an 'X' will be deleted before submission;
they are being classified as unstable at this time.

uchar_t  inq_dtype;          /* Periph. qualifier, dev. type */
uchar_t  inq_rmb       :1;   /* Removable media */
uchar_t  inq_qual      :7;   /* Dev. type qualifier */
uchar_t  inq_iso       :2;   /* ISO version */
uchar_t  inq_ecma      :3;   /* EMCA version */
uchar_t  inq_ansi      :3;   /* ANSII version */
uchar_t  inq_aenc      :1;   /* Async event notif. cap. */
uchar_t  inq_trmiop    :1;   /* Supports TERMINATE I/O PROC msg */
uchar_t  inq_normaca   :1;   /* setting NACA bit supported */
uchar_t  inq_hisup     :1;   /* hierarchical addressing model */
uchar_t  inq_rdf       :4;   /* Response data format */
uchar_t  inq_len             /* Additional length */
uchar_t  inq_sccs      :1;   /* embedded storage array */
Xuchar_t inq_acc       :1;   /* access controls coordinator */
uchar_t  inq_tpgse     :1;   /* explicit asymmetric lun access */
uchar_t  inq_tpgsi     :1;   /* implicit asymmetric lun access */
Xuchar_t inq_3pc       :1;   /* third-party copy */
Xuchar_t inq_protect   :1;   /* supports protection information */
uchar_t  inq_bque      :1;   /* basic queueing */
uchar_t  inq_encserv   :1;   /* embedded enclosure services */
uchar_t  inq_dualp     :1;   /* dual port device */
uchar_t  inq_mchngr    :1;   /* embedded/attached to medium chngr */
uchar_t  inq_addr16    :1;   /* SPI: supports 16 bit wide SCSI addr */
uchar_t  inq_wbus16    :1;   /* SPI: Supports 16 bit wide data xfers */
uchar_t  inq_sync      :1;   /* SPI: Supports synchronous data xfers */
uchar_t  inq_linked    :1;   /* Supports linked commands */
uchar_t  inq_cmd_que   :1;   /* Supports command queueing */
uchar_t  inq_sftre     :1;   /* Supports Soft Reset option */
char     inq_vid[8];         /* Vendor ID */
char     inq_pid[16];        /* Product ID */
char     inq_revision[4];    /* Revision level */
uchar_t  inq_clk       :2;   /* SPI3 clocking */
uchar_t  inq_qas       :1;   /* SPI3: quick arb sel */
uchar_t  inq_ius       :1;   /* SPI3: information units */
```

`inq_dtype` identifies the type of device. Bits 0 - 4 represent the Peripheral Device Type and bits 5 - 7 represent the Peripheral Qualifier. The following values are appropriate for Peripheral Device Type field:

`DTYPE_DIRECT`	Direct-access device (for example, magnetic disk).
`DTYPE_SEQUENTIAL`	Sequential-access device (for example, magnetic tape).
`DTYPE_PRINTER`	Printer device.
`DTYPE_PROCESSOR`	Processor device.
`DTYPE_WORM`	Write-once device (for example, some optical disks).
`DTYPE_RODIRECT`	CD-ROM device.
`DTYPE_SCANNER`	Scanner device.
`DTYPE_OPTICAL`	Optical memory device (for example, some optical disks).
`DTYPE_CHANGER`	Medium Changer device (for example, jukeboxes).
`DTYPE_COMM`	Communications device.
`DTYPE_ARRAY_CTRL`	Array controller device (for example, RAID).
`DTYPE_ESI`	Enclosure services device.
`DTYPE_RBC`	Simplified direct-access device.
`DTYPE_OCRW`	Optical card reader/writer device.
`DTYPE_BRIDGE`	Bridge.
`DTYPE_OSD`	Object-based storage device.
`DTYPE_UNKNOWN`	Unknown or no device type.
`DTYPE_MASK`	Mask to isolate Peripheral Device Type field.

The following values are appropriate for the Peripheral Qualifier field:

`DPQ_POSSIBLE`
: The specified peripheral device type is currently connected to this logical unit. If the target cannot determine whether or not a physical device is currently connected, it uses this peripheral qualifier when returning the `INQUIRY` data. This peripheral qualifier does not imply that the device is ready for access by the initiator.

`DPQ_SUPPORTED`
: The target is capable of supporting the specified peripheral device type on this logical unit. However, the physical device is not currently connected to this logical unit.

`DPQ_NEVER`
: The target is not capable of supporting a physical device on this logical unit. For this peripheral qualifier, the peripheral device type shall be set to

DTYPE_UNKNOWN to provide compatibility with previous versions of SCSI. For all other peripheral device type values, this peripheral qualifier is reserved.

DPQ_VUNIQ — This is a vendor-unique qualifier.

DPQ_MASK — Mask to isolate Peripheral Qualifier field.

DTYPE_NOTPRESENT is the peripheral qualifier DPQ_NEVER and the peripheral device type DTYPE_UNKNOWN combined.

inq_rmb, if set, indicates that the medium is removable.

inq_qual is a device type qualifier.

inq_iso indicates ISO version.

inq_ecma indicates ECMA version.

inq_ansi indicates ANSI version.

inq_aenc, if set, indicates that the device supports asynchronous event notification capability as defined in SCSI-2 specification.

inq_trmiop, if set, indicates that the device supports the TERMINATE I/O PROCESSmessage.

inq_normaca, if set, indicates that the device supports setting the NACA bit to 1 in CDB.

inq_hisip, if set, indicates the SCSI target device uses the hierarchical addressing model to assign LUNs to logical units.

inq_rdf, if set, indicates the INQUIRY data response data format: "RDF_LEVEL0" means that this structure complies with the SCSI-1 spec, "RDF_CCS" means that this structure complies with the CCS pseudo-spec, and "RDF_SCSI2" means that the structure complies with the SCSI-2/3 spec.

inq_len, if set, is the additional length field that specifies the length in bytes of the parameters.

inq_sccs, if set, indicates the target device contains an embedded storage array controller component.

inq_acc, if set, indicates that the logical unit contains an access controls coordinator (this structure member will be deleted before submission. It is being classified as unstable at this time).

inq_tpgse, if set, indicates that implicit asymmetric logical unit access is supported.

inq_tpgsi, if set, indicates that explicit asymmetric logical unit access is supported.

inq_3pc, if set, indicates that the SCSI target device supports third-party copy commands (this structure member will be deleted before submission. It is being classified as unstable at this time).

inq_protect, if set, indicates that the logical unit supports protection information (this structure member will be deleted before submission. It is being classified as unstable at this time).

inq_bque, if set, indicates that the logical unit supports basic task management.

inq_encserv, if set, indicates that the device contains an embedded enclosure services component (ses(7D)).

inq_dualp, if set, indicates that the SCSI target device supports two or more ports.

inq_mchngr, if set, indicates that the SCSI target device supports commands to control an attached media changer.

inq_addr16, if set, indicates that the device supports 16-bit wide SCSI addresses.

inq_wbus16, if set, indicates that the device supports 16-bit wide data transfers.

inq_sync, if set, indicates that the device supports synchronous data transfers.

inq_linked, if set, indicates that the device supports linked commands for this logical unit.

inq_cmdque, if set, indicates that the device supports tagged command queueing.

inq_sftre, if reset, indicates that the device responds to the RESET condition with the hard RESET alternative. If this bit is set, this indicates that the device responds with the soft RESET alternative.

inq_vid contains eight bytes of ASCII data identifying the vendor of the product.

inq_pid contains sixteen bytes of ASCII data as defined by the vendor.

inq_revision contains four bytes of ASCII data as defined by the vendor.

inq_clk clocking of the SPI3 target port.

inq_qas the SPI3 target port supports quick arbitration and selection.

inq_ius the SPI3 target device supports information unit transfers.

See Also scsi_probe(9F), scsi_device(9S)

ANSI Small Computer System Interface-2 (SCSI-2)

ANSI SCSI Primary Commands-3 (SPC-3)

http://t10.org/drafts.htm#spc3

Writing Device Drivers

Name scsi_pkt – SCSI packet structure

Synopsis `#include <sys/scsi/scsi.h>`

Interface Level Solaris DDI specific (Solaris DDI).

Description A `scsi_pkt` structure defines the packet that is allocated by `scsi_init_pkt(9F)`. The target driver fills in some information and passes it to `scsi_transport(9F)` for execution on the target. The host bus adapter (HBA) fills in other information as the command is processed. When the command completes or can be taken no further, the completion function specified in the packet is called with a pointer to the packet as its argument. From fields within the packet, the target driver can determine the success or failure of the command.

Structure Members

```
opaque_t             pkt_ha_private;           /* private data for
                                                  host adapter */
struct scsi_address  pkt_address;              /* destination packet */
opaque_t             pkt_private;              /* private data
                                                  for target driver */
void                 (*pkt_comp)(struct scsi_pkt *); /* callback */
uint_t               pkt_flags;                /* flags */
int                  pkt_time;                 /* time allotted to
                                                  complete command */
uchar_t              *pkt_scbp;                /* pointer to
                                                  status block */
uchar_t              *pkt_cdbp;                /* pointer to
                                                  command block */
ssize_t              pkt_resid;                /* number of bytes
                                                  not transferred */
uint_t               pkt_state;                /* state of command */
uint_t               pkt_statistics;           /* statistics */
uchar_t              pkt_reason;               /* reason completion
                                                  called */
uint_t               pkt_cdblen;               /* length of pkt_cdbp */
uint_t               pkt_scdblen;              /* length of pkt_scbp */
uint_t               pkt_tgtlen;               /* length of pkt_private */
uint_t               pkt_numcookies;           /* number of DMA cookies */
ddi_dma_cookie_t     *pkt_cookies;             /* array of DMA cookies */
uint_t               pkt_dma_flags;            /* DMA flags */
```

`pkt_ha_private` Opaque pointer that the HBA uses to reference a private data structure that transfers `scsi_pkt` requests.

`pkt_address` Initialized by `scsi_init_pkt(9F)`, `pkt_address` records the intended route and the recipient of a request.

`pkt_private` Reserved for the use of the target driver, `pkt_private` is not changed by the HBA driver.

pkt_comp	Specifies the command completion callback routine. When the host adapter driver has gone as far as it can in transporting a command to a SCSI target, and the command has either run to completion or can go no further for some other reason, the host adapter driver calls the function pointed to by this field and passes a pointer to the packet as argument. The callback routine itself is called from interrupt context and must not sleep or call any function that might sleep.
pkt_flags	Provides additional information about how the target driver expects the command to be executed. See pkt_flag Definitions.
pkt_time	Set by the target driver to represent the maximum time allowed in seconds for this command to complete. Timeout starts when the command is transmitted on the SCSI bus. The pkt_time may be 0 if no timeout is required.
pkt_scbp	Points to either a struct scsi_status(9S) or, if auto–rqsense is enabled and pkt_state includes STATE_ARQ_DONE, a struct scsi_arq_status. If scsi_status is returned, the SCSI status byte resulting from the requested command is available. If scsi_arq_status(9S) is returned, the sense information is also available.
pkt_cdbp	Points to a kernel-addressable buffer with a length specified by a call to the proper resource allocation routine, scsi_init_pkt(9F).
pkt_resid	Contains a residual count, either the number of data bytes that have not been transferred (scsi_transport(9F)) or the number of data bytes for which DMA resources could not be allocated scsi_init_pkt(9F). In the latter case, partial DMA resources can be allocated only if scsi_init_pkt(9F) is called with the PKT_DMA_PARTIAL flag.
pkt_state	Has bit positions that represent the six most important states that a SCSI command can go through. See pkt_state Definitions.
pkt_statistics	Maintains some transport-related statistics. See pkt_statistics Definitions.
pkt_reason	Contains a completion code that indicates why the pkt_comp function was called. See pkt_reason Definitions.
pkt_cdblen	Length of buffer pointed to by pkt_cdbp. See tran_setup_pkt.
pkt_scblen	Length of buffer pointed to by pkt_scbp. See tran_setup_pkt.
pkt_tgtlen	Length of buffer pointed to by pkt_private. See tran_setup_pkt.
pkt_numcookies	Length pkt_cookies array. See tran_setup_pkt.
pkt_cookies	Array of DMA cookies. See tran_setup_pkt.

`pkt_dma_flags`	DMA flags used, such as `DDI_DMA_READ` and `DDI_DMA_WRITE`. See `tran_setup_pkt`.

The host adapter driver will update the `pkt_resid`, `pkt_reason`, `pkt_state`, and `pkt_statistics` fields.

`pkt_flags` Definitions

The appropriate definitions for the structure member `pkt_flags` are:

`FLAG_NOINTR`	Run command with no command completion callback. Command is complete upon return from `scsi_transport`(9F).
`FLAG_NODISCON`	Run command without disconnects.
`FLAG_NOPARITY`	Run command without parity checking.
`FLAG_HTAG`	Run command as the head-of-queue-tagged command.
`FLAG_OTAG`	Run command as an ordered-queue-tagged command.
`FLAG_STAG`	Run command as a simple-queue-tagged command.
`FLAG_SENSING`	Indicates a request sense command.
`FLAG_HEAD`	Place command at the head of the queue.
`FLAG_RENEGOTIATE_WIDE_SYNC`	Before transporting this command, the host adapter should initiate the renegotiation of wide mode and synchronous transfer speed. Normally, the HBA driver manages negotiations but under certain conditions forcing a renegotiation is appropriate. Renegotiation is recommended before Request Sense and Inquiry commands. Refer to the SCSI 2 standard, sections 6.6.21 and 6.6.23. This flag should not be set for every packet as this will severely impact performance.

`pkt_reason` Definitions

The appropriate definitions for the structure member `pkt_reason` are:

`CMD_CMPLT`	No transport errors; normal completion.
`CMD_INCOMPLETE`	Transport stopped with abnormal state.
`CMD_DMA_DERR`	DMAd irection error.
`CMD_TRAN_ERR`	Unspecified transport error.
`CMD_RESET`	SCSI bus reset destroyed command.
`CMD_ABORTED`	Command transport aborted on request.

CMD_TIMEOUT	Command timed out.
CMD_DATA_OVR	Data overrun.
CMD_CMD_OVR	Command overrun.
CMD_STS_OVR	Status overrun.
CMD_BADMSG	Message not command complete.
CMD_NOMSGOUT	Target refused to go to message out phase.
CMD_XID_FAIL	Extended identify message rejected.
CMD_IDE_FAIL	"Initiator Detected Error" message rejected.
CMD_ABORT_FAIL	Abort message rejected.
CMD_REJECT_FAIL	Reject message rejected.
CMD_NOP_FAIL	"No Operation" message rejected.
CMD_PER_FAIL	"Message Parity Error" message rejected.
CMD_BDR_FAIL	"Bus Device Reset" message rejected.
CMD_ID_FAIL	Identify message rejected.
CMD_UNX_BUS_FREE	Unexpected bus free phase.
CMD_TAG_REJECT	Target rejected the tag message.
CMD_DEV_GONE	The device has been removed.

pkt_state Definitions

The appropriate definitions for the structure member `pkt_state` are:

STATE_GOT_BUS	Bus arbitration succeeded.
STATE_GOT_TARGET	Target successfully selected.
STATE_SENT_CMD	Command successfully sent.
STATE_XFERRED_DATA	Data transfer took place.
STATE_GOT_STATUS	Status received.
STATE_ARQ_DONE	The command resulted in a check condition and the host adapter driver executed an automatic request sense command.

pkt_statistics Definitions

The definitions that are appropriate for the structure member `pkt_statistics` are:

STAT_DISCON	Device disconnect.
STAT_SYNC	Command did a synchronous data transfer.
STAT_PERR	SCSI parity error.

STAT_BUS_RESET	Bus reset.
STAT_DEV_RESET	Device reset.
STAT_ABORTED	Command was aborted.
STAT_TIMEOUT	Command timed out.

See Also tran_init_pkt(9E), scsi_arq_status(9S), scsi_init_pkt(9F), scsi_status(9S), scsi_hba_pkt_comp(9F)

Writing Device Drivers

Notes HBA drivers should signal scsi_pkt completion by calling scsi_hba_pkt_comp(9F). This is mandatory for HBA drivers that implement tran_setup_pkt. Failure to comply results in undefined behavior.

Name scsi_status – SCSI status structure

Synopsis `#include <sys/scsi/scsi.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description The SCSI-2standard defines a status byte that is normally sent by the target to the initiator during the status phase at the completion of each command.

Structure Members

```
uchar  sts_scsi2    :1;      /* SCSI-2 modifier bit */
uchar  sts_is       :1;      /* intermediate status sent */
uchar  sts_busy     :1;      /* device busy or reserved */
uchar  sts_cm       :1;      /* condition met */
ucha   sts_chk      :1;      /* check condition */
```

`sts_chk` indicates that a contingent allegiance condition has occurred.

`sts_cm` is returned whenever the requested operation is satisfied

`sts_busy` indicates that the target is busy. This status is returned whenever a target is unable to accept a command from an otherwise acceptable initiator (that is, no reservation conflicts). The recommended initiator recovery action is to issue the command again later.

`sts_is` is returned for every successfully completed command in a series of linked commands (except the last command), unless the command is terminated with a check condition status, reservation conflict, or command terminated status. Note that host bus adapter drivers may not support linked commands (see `scsi_ifsetcap`(9F)). If `sts_is` and `sts_busy` are both set, then a reservation conflict has occurred.

`sts_scsi2` is the SCSI-2 modifier bit. If `sts_scsi2` and `sts_chk` are both set, this indicates a command terminated status. If `sts_scsi2` and `sts_busy` are both set, this indicates that the command queue in the target is full.

For accessing the status as a byte, the following values are appropriate:

`STATUS_GOOD` This status indicates that the target has successfully completed the command.

`STATUS_CHECK` This status indicates that a contingent allegiance condition has occurred.

`STATUS_MET` This status is returned when the requested operations are satisfied.

`STATUS_BUSY` This status indicates that the target is busy.

`STATUS_INTERMEDIATE` This status is returned for every successfully completed command in a series of linked commands.

`STATUS_SCSI2` This is the SCSI-2 modifier bit.

STATUS_INTERMEDIATE_MET	This status is a combination of STATUS_MET and STATUS_INTERMEDIATE.
STATUS_RESERVATION_CONFLICT	This status is a combination of STATUS_INTERMEDIATE and STATUS_BUSY, and it is returned whenever an initiator attempts to access a logical unit or an extent within a logical unit is reserved.
STATUS_TERMINATED	This status is a combination of STATUS_SCSI2 and STATUS_CHECK, and it is returned whenever the target terminates the current I/O process after receiving a terminate I/O process message.
STATUS_QFULL	This status is a combination of STATUS_SCSI2 and STATUS_BUSY, and it is returned when the command queue in the target is full.

See Also scsi_ifgetcap(9F), scsi_init_pkt(9F), scsi_extended_sense(9S), scsi_pkt(9S)

Writing Device Drivers

Name streamtab – STREAMS entity declaration structure

Synopsis

```
#include <sys/stream.h>
```

Interface Level Architecture independent level 1 (DDI/DKI).

Description Each STREAMS driver or module must have a `streamtab` structure.

`streamtab` is made up of `qinit` structures for both the read and write queue portions of each module or driver. Multiplexing drivers require both upper and lower `qinit` structures. The `qinit` structure contains the entry points through which the module or driver routines are called.

Normally, the read `QUEUE` contains the `open` and `close` routines. Both the read and write queue can contain `put` and service procedures.

Structure Members

```
struct qinit    *st_rdinit;    /* read QUEUE */
struct qinit    *st_wrinit;    /* write QUEUE */
struct qinit    *st_muxrinit;  /* lower read QUEUE*/
struct qinit    *st_muxwinit;  /* lower write QUEUE*/
```

See Also `qinit(9S)`

STREAMS Programming Guide

Name stroptions – options structure for M_SETOPTS message

Synopsis

```
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

Interface Level Architecture independent level 1 (DDI/DKI)

Description The M_SETOPTS message contains a stroptions structure and is used to control options in the stream head.

Structure Members

```
uint_t          so_flags;      /* options to set */
short           so_readopt;    /* read option */
ushort_t        so_wroff;      /* write offset */
ssize_t         so_minpsz;     /* minimum read packet size */
ssize_t         so_maxpsz;     /* maximum read packet size */
size_t          so_hiwat;      /* read queue high water mark */
size_t          so_lowat;      /* read queue low water mark */
unsigned char   so_band;       /* band for water marks */
ushort_t        so_erropt;     /* error option */
```

The following are the flags that can be set in the so_flags bit mask in the stroptions structure. Note that multiple flags can be set.

SO_READOPT	Set read option.
SO_WROFF	Set write offset.
SO_MINPSZ	Set minimum packet size
SO_MAXPSZ	Set maximum packet size.
SO_HIWAT	Set high water mark.
SO_LOWAT	Set low water mark.
SO_MREADON	Set read notification ON.
SO_MREADOFF	Set read notification OFF.
SO_NDELON	Old TTY semantics for NDELAY reads and writes.
SO_NDELOFFSTREAMS	Semantics for NDELAY reads and writes.
SO_ISTTY	The stream is acting as a terminal.
SO_ISNTTY	The stream is not acting as a terminal.
SO_TOSTOP	Stop on background writes to this stream.
SO_TONSTOP	Do not stop on background writes to this stream.
SO_BAND	Water marks affect band.

SO_ERROPT	Set error option.

When SO_READOPT is set, the so_readopt field of the stroptions structure can take one of the following values. See read(2).

RNORM	Read message normal.
RMSGD	Read message discard.
RMSGN	Read message, no discard.

When SO_BAND is set, so_band determines to which band so_hiwat and so_lowat apply.

When SO_ERROPT is set, the so_erropt field of the stroptions structure can take a value that is either none or one of:

RERRNORM	Persistent read errors; default.
RERRNONPERSIST	Non-persistent read errors.

OR'ed with either none or one of:

WERRNORM	Persistent write errors; default.
WERRNONPERSIST	Non-persistent write errors.

See Also

read(2), streamio(7I)

STREAMS Programming Guide

Name tuple – card information structure (CIS) access structure

Synopsis `#include <sys/pccard.h>`

Interface Level Solaris DDI Specific (Solaris DDI)

Description The `tuple_t` structure is the basic data structure provided by card services to manage PC card information. A PC card provides identification and configuration information through its card information structure (CIS). A PC card driver accesses a PC card's CIS through various card services functions.

The CIS information allows PC cards to be self-identifying: the CIS provides information to the system so that it can identify the proper PC card driver for the PC card, and provides configuration information so that the driver can allocate appropriate resources to configure the PC card for proper operation in the system.

The CIS information is contained on the PC card in a linked list of tuple data structures called a CIS chain. Each tuple has a one-byte type and a one-byte link, an offset to the next tuple in the list. A PC card can have one or more CIS chains.

A multi-function PC card that complies with the PC Card 95 MultiFunction Metaformat specification will have one or more global CIS chains that collectively are referred to as the global CIS. These PC Cards will also have one or more per-function CIS chains. Each per-function collection of CIS chains is referred to as a function-specific CIS.

To examine a PC card's CIS, first a PC card driver must locate the desired tuple by calling `csx_GetFirstTuple(9F)`. Once the first tuple is located, subsequent tuples may be located by calling `csx_GetNextTuple(9F)`. See `csx_GetFirstTuple(9F)`. The linked list of tuples may be inspected one by one, or the driver may narrow the search by requesting only tuples of a particular type.

Once a tuple has been located, the PC card driver may inspect the tuple data. The most convenient way to do this for standard tuples is by calling one of the number of tuple-parsing utility functions; for custom tuples, the driver may get access to the raw tuple data by calling `csx_GetTupleData(9F)`.

Solaris PC card drivers do not need to be concerned with which CIS chain a tuple appears in. On a multi-function PC card, the client will get the tuples from the global CIS followed by the tuples in the function-specific CIS. The caller will not get any tuples from a function-specific CIS that does not belong to the caller's function.

Structure Members The structure members of `tuple_t` are:

```
uint32_t      Socket;          /* socket number */
uint32_t      Attributes;      /* tuple attributes */
cisdata_t     DesiredTuple;    /* tuple to search for */
cisdata_t     TupleOffset;     /* tuple data offset */
```

```
cisdata_t     TupleDataMax;    /* max tuple data size */
cisdata_t     TupleDataLen;    /* actual tuple data length */
cisdata_t     TupleData[CIS_MAX_TUPLE_DATA_LEN];
                               /* body tuple data */
cisdata_t     TupleCode;       /* tuple type code */
cisdata_t     TupleLink;       /* tuple link */
```

The fields are defined as follows:

`Socket` Not used in Solaris, but for portability with other card services implementations, it should be set to the logical socket number.

`Attributes` This field is bit-mapped. The following bits are defined:

`TUPLE_RETURN_LINK`	Return link tuples if set.
`TUPLE_RETURN_IGNORED_TUPLES`	Return ignored tuples if set. Ignored tuples are those tuples in a multi-function PC card's global CIS chain that are duplicates of the same tuples in a function-specific CIS chain.
`TUPLE_RETURN_NAME`	Return tuple name string using the `csx_ParseTuple(9F)` function if set.

`DesiredTuple` This field is the requested tuple type code to be returned when calling `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`. `RETURN_FIRST_TUPLE` is used to return the first tuple regardless of tuple type. `RETURN_NEXT_TUPLE` is used to return the next tuple regardless of tuple type.

`TupleOffset` This field allows partial tuple information to be retrieved, starting at the specified offset within the tuple. This field must only be set before calling `csx_GetTupleData(9F)`.

`TupleDataMax` This field is the size of the tuple data buffer that card services uses to return raw tuple data from `csx_GetTupleData(9F)`. It can be larger than the number of bytes in the tuple data body. Card services ignores any value placed here by the client.

`TupleDataLen` This field is the actual size of the tuple data body. It represents the number of tuple data body bytes returned by `csx_GetTupleData(9F)`.

`TupleData` This field is an array of bytes containing the raw tuple data body contents returned by `csx_GetTupleData(9F)`.

`TupleCode` This field is the tuple type code and is returned by `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)` when a tuple matching the `DesiredTuple` field is returned.

`TupleLink` This field is the tuple link, the offset to the next tuple, and is returned by `csx_GetFirstTuple`(9F) or `csx_GetNextTuple`(9F) when a tuple matching the `DesiredTuple` field is returned.

See Also `csx_GetFirstTuple`(9F), `csx_GetTupleData`(9F), `csx_ParseTuple`(9F), `csx_Parse_CISTPL_BATTERY`(9F), `csx_Parse_CISTPL_BYTEORDER`(9F), `csx_Parse_CISTPL_CFTABLE_ENTRY`(9F), `csx_Parse_CISTPL_CONFIG`(9F), `csx_Parse_CISTPL_DATE`(9F), `csx_Parse_CISTPL_DEVICE`(9F), `csx_Parse_CISTPL_FUNCE`(9F), `csx_Parse_CISTPL_FUNCID`(9F), `csx_Parse_CISTPL_JEDEC_C`(9F), `csx_Parse_CISTPL_MANFID`(9F), `csx_Parse_CISTPL_SPCL`(9F), `csx_Parse_CISTPL_VERS_1`(9F), `csx_Parse_CISTPL_VERS_2`(9F)

PC Card 95 Standard, PCMCIA/JEIDA

Name uio – scatter/gather I/O request structure

Synopsis `#include <sys/uio.h>`

Interface Level Architecture independent level 1 (DDI/DKI)

Description A `uio` structure describes an I/O request that can be broken up into different data storage areas (scatter/gather I/O). A request is a list of `iovec` structures (base-length pairs) indicating where in user space or kernel space the I/O data is to be read or written.

The contents of `uio` structures passed to the driver through the entry points should not be written by the driver. The `uiomove(9F)` function takes care of all overhead related to maintaining the state of the `uio` structure.

`uio` structures allocated by the driver should be initialized to zero before use, by `bzero(9F)`, `kmem_zalloc(9F)`, or an equivalent.

Structure Members

```
iovec_t      *uio_iov;      /* pointer to start of iovec */
                            /* list for uio struc. */
int          uio_iovcnt;    /* number of iovecs in list */
off_t        uio_offset;    /* 32-bit offset into file where
                            /* data is xferred. See NOTES. */
offset_t     uio_loffset;   /* 64-bit offset into file where */
                            /* data is xferred. See NOTES. */
uio_seg_t    uio_segflg;    /* ID's type of I/O transfer: */
                            /* UIO_SYSSPACE:  kernel <-> kernel */
                            /* UIO_USERSPACE: kernel <-> user */
short        uio_fmode;     /* file mode flags (not driver setable) */
daddr_t      uio_limit;     /* 32-bit ulimit for file (max. block */
                            /* offset). not driver setable. */
                            /* See NOTES. */
diskaddr_t   uio_llimit;    /* 64-bit ulimit for file (max. block */
                            /* offset). not driver setable. */
                            /* See NOTES */
int          uio_resid;     /* residual count */
```

The `uio_iov` member is a pointer to the beginning of the `iovec(9S)` list for the `uio`. When the `uio` structure is passed to the driver through an entry point, the driver should not set `uio_iov`. When the `uio` structure is created by the driver, `uio_iov` should be initialized by the driver and not written to afterward.

See Also `aread(9E)`, `awrite(9E)`, `read(9E)`, `write(9E)`, `bzero(9F)`, `kmem_zalloc(9F)`, `uiomove(9F)`, `cb_ops(9S)`, `iovec(9S)`

Writing Device Drivers

Notes Only one structure, uio_offset or uio_loffset, should be interpreted by the driver. Which field the driver interprets is dependent upon the settings in the cb_ops(9S) structure.

Only one structure, uio_limit or uio_llimit, should be interpreted by the driver. Which field the driver interprets is dependent upon the settings in the cb_ops(9S) structure.

When performing I/O on a seekable device, the driver should not modify either the uio_offset or the uio_loffset field of the uio structure. I/O to such a device is constrained by the maximum offset value. When performing I/O on a device on which the concept of position has no relevance, the driver may preserve the uio_offset or uio_loffset, perform the I/O operation, then restore the uio_offset or uio_loffset to the field's initial value. I/O performed to a device in this manner is not constrained.

Name usb_bulk_request – USB bulk request structure

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description A bulk request (that is, a request sent through a bulk pipe) is used to transfer large amounts of data in reliable but non-time-critical fashion. Please refer to Section *5.8* of the *USB 2.0* specification for information on bulk transfers. (The *USB 2.0* specification is available at *www.usb.org*.)

The fields in the usb_bulk_req_t are used to format a bulk request. Please see below for acceptable combinations of flags and attributes.

The usb_bulk_req_t fields are:

```
uint_t          bulk_len;       /* number of bytes to xfer      */
                                /* Please see */
                                /* usb_pipe_get_max_bulk_xfer_size(9F) */
                                /* for maximum size */
mblk_t          *bulk_data;     /* the data for the data phase  */
                                /* IN or OUT: allocated by client */
uint_t          bulk_timeout;   /* xfer timeout value in secs   */
                                /* If set to zero, defaults to 5 sec */
usb_opaque_t    bulk_client_private; /* Client specific information */
usb_req_attrs_t bulk_attributes; /* xfer-attributes     */

/* Normal callback function, called upon completion. */
void            (*bulk_cb)(
                    usb_pipe_handle_t ph, struct usb_bulk_req *req);

/* Exception callback function, for error handling. */
void            (*bulk_exc_cb)(
                    usb_pipe_handle_t ph, struct usb_bulk_req *req);

/* set by USBA/HCD framework on completion */
usb_cr_t        bulk_completion_reason; /* overall success status */
                                   /* See usb_completion_reason(9S) */
usb_cb_flags_t  bulk_cb_flags; /* recovery done by callback hndlr */
                                   /* See usb_callback_flags(9S) */
```

Request attributes define special handling for transfers. The following attributes are valid for bulk requests:

USB_ATTRS_SHORT_XFER_OK — USB framework accepts transfers where less data is received than expected.

USB_ATTRS_AUTOCLEARING — USB framework resets pipe and clears functional stalls automatically on exception.

USB_ATTRS_PIPE_RESET	USB framework resets pipe automatically on exception.

Please see `usb_request_attributes(9S)` for more information.

Bulk transfers/requests are subject to the following constraints and caveats:

1) The following table indicates combinations of `usb_pipe_bulk_xfer()` flags argument and fields of the usb_bulk_req_t request argument (X = don't care).

```
Flags     Type  Attributes       Data   Timeout Semantics
---------------------------------------------------------------
 X         X     X                ==NULL X      illegal

 X         X     ONE_XFER         X      X      illegal

 no sleep  IN    !SHORT_XFER_OK   !=NULL 0      See  note (A)

 no sleep  IN    !SHORT_XFER_OK   !=NULL > 0    See  note (B)

 sleep     IN    !SHORT_XFER_OK   !=NULL 0      See  note (C)

 sleep     IN    !SHORT_XFER_OK   !=NULL > 0    See  note (D)

 no sleep  IN    SHORT_XFER_OK    !=NULL 0      See  note (E)

 no sleep  IN    SHORT_XFER_OK    !=NULL > 0    See  note (F)

 sleep     IN    SHORT_XFER_OK    !=NULL 0      See  note (G)

 sleep     IN    SHORT_XFER_OK    !=NULL > 0    See  note (H)

 X         OUT   SHORT_XFER_OK   X       X      illegal

 no sleep  OUT   X               !=NULL  0      See  note (I)

 no sleep  OUT   X               !=NULL  > 0    See  note (J)

 sleep     OUT   X               !=NULL  0      See  note (K)

 sleep     OUT   X               !=NULL  > 0    See  note (L)
```

Table notes:

A). Fill buffer, no timeout, callback when bulk_len is transferred.
B). Fill buffer, with timeout; callback when bulk_len is transferred.
C). Fill buffer, no timeout, unblock when bulk_len is transferred; no callback.

D). Fill buffer, with timeout; unblock when bulk_len is transferred or a timeout occurs; no callback.
E) Fill buffer, no timeout, callback when bulk_len is transferred or first short packet is received.
F). Fill buffer, with timeout; callback when bulk_len is transferred or first short packet is received.
G). Fill buffer, no timeout, unblock when bulk_len is transferred or first short packet is received; no callback.
H). Fill buffer, with timeout; unblock when bulk_len is transferred, first short packet is received, or a timeout occurs; no callback.
I). Empty buffer, no timeout; callback when bulk_len is transferred.
J) Empty buffer, with timeout; callback when bulk_len is transferred or a timeout occurs.
K). Empty buffer, no timeout; unblock when bulk_len is transferred; no callback.
L). Empty buffer, with timeout; unblock when bulk_len is transferred or a timeout occurs; no callback.

2) bulk_len must be > 0. bulk_data must not be NULL.

3) Bulk_residue is set for both READ and WRITE. If it is set to 0, it means that all of the data was transferred successfully. In case of WRITE it contains data not written and in case of READ it contains the data NOT read so far. A residue can only occur because of timeout or bus/device error. (Note that a short transfer for a request where the USB_ATTRS_SHORT_XFER_OK attribute is not set is considered a device error.) An exception callback is made and completion_reason will be non-zero.

4) Splitting large Bulk xfers: Due to internal constraints, the USBA framework can only do a limited size bulk data xfer per request. A client driver may first determine this limitation by calling the USBA interface (usb_pipe_get_max_bulk_xfer_size(9F)) and then restrict itself to doing transfers in multiples of this fixed size. This forces a client driver to do data xfers in a loop for a large request, splitting it into multiple chunks of fixed size.

The bulk_completion_reason indicates the status of the transfer. See `usb_completion_reason(9S)` for usb_cr_t definitions.

The bulk_cb_flags are set prior to calling the exception callback handler to summarize recovery actions taken and errors encountered during recovery. See `usb_callback_flags(9S)` for usb_cb_flags_t definitions.

--- Callback handling ---

All usb request types share the same callback handling. See `usb_callback_flags(9S)` for details.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also usb_alloc_request(9F), usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_get_max_bulk_transfer_size(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_callback_flags(9S), usb_completion_reason(9S), usb_ctrl_request(9S), usb_intr_request(9S), usb_isoc_request(9S), usb_request_attributes(9S)

Name usb_callback_flags – USB callback flag definitions

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description If the USB framework detects an error during a request execution, it calls the client driver's exception callback handler to report what happened. Callback flags (which are set prior to calling the exception callback handler) detail errors discovered during the exception recovery process, and summarize recovery actions taken by the USBA framework.

Information from the callback flags supplements information from the original transport error. For transfers, the original transport error status is returned to the callback handler through the original request (whose completion reason field contains any transport error indication). For command completion callbacks, the callback's rval argument contains the transport error status. A completion reason of `USB_CR_OK` means the transfer completed with no errors detected.

The usb_cb_flags_t enumerated type contains the following definitions:

USB_CB_NO_INFO
: No additional errors discovered or recovery actions taken.

USB_CB_FUNCTIONAL_STALL
: A functional stall occurred during the transfer. A functional stall is usually caused by a hardware error, and must be explicitly cleared. A functional stall is fatal if it cannot be cleared. The default control pipe never shows a functional stall.

USB_CB_STALL_CLEARED
: A functional stall has been cleared by the USBA framework. This can happen if USB_ATTRS_AUTOCLEARING is set in the request's xxxx_attributes field.

USB_CB_PROTOCOL_STALL
: A protocol stall has occurred during the transfer. A protocol stall is caused usually by an invalid or misunderstood command. It is cleared automatically when the device is given its next command. The USBA framework treats stalls detected on default pipe transfers as protocol stalls.

USB_CB_RESET_PIPE
: A pipe with a stall has been reset automatically via autoclearing, or via an explicit call to `usb_pipe_reset(9F)`. Resetting a pipe consists of stopping all transactions on a pipe, setting the pipe to the idle state, and if the pipe is not the default pipe, flushing all pending requests. The request which has the error, plus all pending requests which are flushed,

	show USB_CB_RESET_PIPE set in the usb_cb_flags_t when their exception callback is called.
USB_CB_ASYNC_REQ_FAILED	Resources could not be allocated to process callbacks asynchronously. Callbacks receiving this flag must not block, since those callbacks are executing in a context which holds resources shared by the rest of the system. Note that exception callbacks with USB_CB_ASYNC_REQ_FAILED set may execute out of order from the requests which preceded them. Normal callbacks may be already queued when an exception hits that the USBA is unable to queue.
USB_CB_SUBMIT_FAILED	A queued request was submitted to the host controller driver and was rejected. The usb_completion_reason shows why the request was rejected by the host controller.
USB_CB_NO_RESOURCES	Insufficient resources were available for recovery to proceed.
USB_CB_INTR_CONTEXT	Callback is executing in interrupt context and should not block.

The usb_cb_flags_t enumerated type defines a bitmask. Multiple bits can be set, reporting back multiple statuses to the exception callback handler.

CALLBACK HANDLER

The USBA framework supports callback handling as a way of asynchronous client driver notification. There are three kinds of callbacks: Normal completion transfer callback, exception (error) completion transfer callback, and command completion callback, each described below.

Callback handlers are called whenever they are specified in a request or command, regardless of whether or not that request or command specifies the USB_FLAGS_SLEEP flag. (USB_FLAGS_SLEEP tells the request or command to block until completed.) Callback handlers must be specified whenever an asynchronous transfer is requested.

PIPE POLICY

Each pipe is associated with a pool of threads that are used to run callbacks associated with requests on that pipe. All transfer completion callbacks for a particular pipe are run serially by a single thread.

Pipes taking requests with callbacks which can block must have their pipe policy properly initialized. If a callback blocks on a condition that is only met by another thread associated with the same pipe, there must be sufficient threads available. Otherwise that callback thread will block forever. Similarly, problems will ensue when callbacks overlap and there are not enough threads to handle the number of overlapping callbacks.

The pp_max_async_reqs field of the pipe_policy provides a hint of how many threads to allocate for asynchronous processing of request callbacks on a pipe. Set this value high enough per pipe to accommodate all of the pipe's possible asynchronous conditions. The pipe_policy is passed to usb_pipe_open(9F).

Transfer completion callbacks (normal completion and exception):

Most transfer completion callbacks are allowed to block, but only under certain conditions:

1. No callback is allowed to block if the callback flags show USB_CB_INTR_CONTEXT set, since that flag indicates that the callback is running in interrupt context instead of kernel context. Isochronous normal completion callbacks, plus those with USB_CB_ASYNC_REQ_FAILED set, execute in interrupt context.
2. Any callback except for isochronous normal completion can block for resources (for example to allocate memory).
3. No callback can block for synchronous completion of a command (for example, a call to usb_pipe_close(9F) with the USB_FLAGS_SLEEP flag passed) done on the same pipe. The command could wait for all callbacks to complete, including the callback which issued that command, causing all operations on the pipe to deadlock. Note that asynchronous commands can start from a callback, providing that the pipe's policy pp_max_async_reqs field is initialized to accommodate them.
4. Avoid callbacks that block for synchronous completion of commands done on other pipes. Such conditions can cause complex dependencies and unpredictable results.
5. No callback can block waiting for a synchronous transfer request to complete. (Note that making an asynchronous request to start a new transfer or start polling does not block, and is OK.)
6. No callback can block waiting for another callback to complete. (This is because all callbacks are done by a single thread.)
7. Note that if a callback blocks, other callbacks awaiting processing can backup behind it, impacting system resources.

A transfer request can specify a non-null normal-completion callback. Such requests conclude by calling the normal-completion callback when the transfer completes normally. Similarly, a transfer request can specify a non-null exception callback. Such requests conclude by calling the exception callback when the transfer completes abnormally. Note that the same callback can be used for both normal completion and exception callback handling. A completion reason of USB_CR_OK defines normal completion.

All request-callbacks take as arguments a usb_pipe_handle_t and a pointer to the request:

```
xxxx_cb(usb_pipe_handle_t ph, struct usb_ctrl_req *req);
```

Such callbacks can retrieve saved state or other information from the private area of the pipe handle. (See usb_pipe_set_private(9F).) Handlers also have access to the completion reason (usb_cr_t) and callback flags (usb_cb_flags_t) through the request argument they are passed.

Request information follows. In the data below, *xxxx* below represents the type of request (ctrl, intr, isoc or bulk.)

```
Request structure name is usb_xxxx_req_t.

Normal completion callback handler field is xxxx_cb.

Exception callback handler field is xxxx_exc_cb.

Completion reason field is xxxx_completion_reason.

Callback flags field is xxxx_cb_flags.
```

COMMAND COMPLETION CALLBACKS

Calls to some non-transfer functions can be set up for callback notification. These include usb_pipe_close(9F), usb_pipe_reset(9F), usb_pipe_drain_reqs(9F), usb_set_cfg(9F), usb_set_alt_if(9F) and usb_clr_feature(9F).

The signature of a command completion callback is as follows:

```
command_cb(
    usb_pipe_handle_t cb_pipe_handle,
    usb_opaque_t arg,
    int rval,
    usb_cb_flags_t flags);
```

As with transfer completion callbacks, command completion callbacks take a usb_pipe_handle_t to retrieve saved state or other information from the pipe's private area. Also, command completion callbacks are provided with an additional user-definable argument (usb_opaque_t arg), the return status of the executed command (int rval), and the callback flags (usb_cb_flags_t flags).

The rval argument is roughly equivalent to the completion reason of a transfer callback, indicating the overall status. See the return values of the relevant function for possible rval values which can be passed to the callback.

The callback flags can be checked when rval indicates failure status. Just as for transfer completion callbacks, callback flags return additional information on execution events.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving
Availability	SUNWusb, SUNWusbu

See Also usb_alloc_request(9F), usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_bulk_request(9S), usb_ctrl_request(9S), usb_intr_request(9S), usb_isoc_request(9S)

Name usb_cfg_descr – USB configuration descriptor

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description The `usb_cfg_descr_t` configuration descriptor defines attributes of a configuration. A configuration contains one or more interfaces. A configuration descriptor acts as a header for the group of other descriptors describing the subcomponents (for example, interfaces and endpoints) of a configuration. Please refer to Section *9.6.3* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

One or more configuration descriptors are retrieved from a USB device during device enumeration. They can be accessed via `usb_get_dev_data(9F)`.

A configuration descriptor has the following fields:

```
uint8_t         bLength                 Size of this descriptor
                                        in bytes.

uint8_t         bDescriptorType         Set to USB_DESCR_TYPE_CFG.

uint16_t        wTotalLength            Total length of data returned
                                        including this and all other
                                        descriptors in this configuration.

uint8_t         bNumInterfaces          Number of interfaces in this
                                        configuration.

uint8_t         bConfigurationValue     ID of this configuration
                                        (1-based).

uint8_t         iConfiguration          Index of optional configuration
                                        string. Valid if > 0.

uint8_t         bmAttributes            Configuration characteristics
                                        (See below).

uint8_t         bMaxPower               Maximum power consumption, in
                                        2mA units.

Configuration descriptors define the following bmAttributes:
     USB_CFG_ATTR_SELFPWR       -      Set if config not using bus power.
     USB_CFG_ATTR_REMOTE_WAKEUP -      Set if config supports rem wakeup.
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also attributes(5), usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_parse_data(9F), usb_ctrl_request(9S), usb_dev_descr(9S), usb_dev_qlf_descr(9S), usb_ep_descr(9S), usb_if_descr(9S), usb_other_speed_cfg_descr(9S), usb_string_descr(9S)

Name usb_client_dev_data – Device configuration information

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The usb_client_dev_data_t structure carries all device configuration information. It is provided to a USB client driver through a call to usb_get_dev_data(9F). Most USBA functions require information which comes from this structure.

The usb_client_dev_data_t structure fields are:

```
usb_pipe_handle_t   dev_default_ph;   /* deflt ctrl pipe handle */

ddi_iblock_cookie_t dev_iblock_cookie;/* for calls to mutex_init */
                                       /* for mutexes used by intr */
                                       /* context callbacks. */

usb_dev_descr_t     *dev_descr;       /* parsed* dev. descriptor */

char                *dev_mfg;         /* manufacturer's ID string */

char                *dev_product;     /* product ID string */

char                *dev_serial;      /* serial num. string */

usb_reg_parse_lvl_t dev_parse_level; /* Parse level */
                                      /* reflecting the tree */
                                      /* (if any) returned through */
                                      /* the dev_cfg array. */

usb_cfg_data_t      *dev_cfg;         /* parsed* descr  tree.*/

uint_t              dev_n_cfg;        /* num cfgs in parsed descr. */
                                      /* tree, dev_cfg array below.*/

usb_cfg_data_t      *dev_curr_cfg;    /* Pointer to the tree config*/
                                      /* corresponding to the cfg */
                                      /* active at the time of the */
                                      /* usb_get_dev_data() call */

int                 dev_curr_if;      /* First active interface in */
                                      /* tree under driver's control.*/

                                      /* Always zero when driver */
                                      /* controls whole device. */
```

```
* A parsed descriptor is in a struct whose fields' have been adjusted
  to the host processor.  This may include endianness adjustment (the USB
```

```
standard defines that devices report in little-endian bit order) or
structure padding as necessary.
```

dev_parse_level represents the extent of the device represented by the tree returned by the dev_cfg field and has the following possible values:

USB_PARSE_LVL_NONE	Build no tree. dev_n_cfg returns 0, dev_cfg and dev_curr_cfg are returned NULL, the dev_curr_xxx fields are invalid.
USB_PARSE_LVL_IF	Parse configured interface only, if configuration# and interface properties are set (as when different interfaces are viewed by the OS as different device instances). If an OS device instance is set up to represent an entire physical device, this works like USB_PARSE_LVL_ALL.
USB_PARSE_LVL_CFG	Parse entire configuration of configured interface only. This is like USB_PARSE_LVL_IF except entire configuration is returned.
USB_PARSE_LVL_ALL	Parse entire device (all configurations), even when driver is bound to a single interface of a single configuration.

The default control pipe handle is used mainly for control commands and device setup.

The dev_iblock_cookie is used to initialize client driver mutexes which are used in interrupt-context callback handlers. (All callback handlers called with USB_CB_INTR_CONTEXT in their usb_cb_flags_t arg execute in interrupt context.) This cookie is used in lieu of one returned by `ddi_get_iblock_cookie(9F)`. Mutexes used in other handlers or under other conditions should initialize per `mutex_init(9F)`.

The parsed standard USB device descriptor is used for device type identification.

The several ID strings, including the manufacturer's ID, product ID, and serial number may be used to identify the device in messages or to compare it to other devices.

The descriptor tree, returned by dev_cfg, makes a device's parsed standard USB descriptors available to the driver. The tree is designed to be easily traversed to get any or all standard *USB 2.0* descriptors. (See the "Tree Structure" section of this manpage below.) dev_n_cfg returns the number of configurations in the tree. Note that this value may differ from the number of configurations returned in the device descriptor.

A returned parse_level field of USB_PARSE_LVL_ALL indicates that all configurations are represented in the tree. This results when USB_PARSE_LVL_ALL is explicitly requested by the caller in the flags argument to `usb_get_dev_data()`, or when the whole device is seen by the system for the current OS device node (as opposed to only a single configuration for that OS device node). USB_PARSE_LVL_CFG is returned when one entire configuration is returned in the tree. USB_PARSE_LVL_IF is returned when one interface of one configuration is returned in the tree. In the latter two cases, the returned configuration is at dev_cfg[USB_DEV_DEFAULT_CONFIG_INDEX]. USB_PARSE_LVL_NONE is returned when no tree is returned. Note that the value of this field can differ from the parse_level requested as an argument to `usb_get_dev_data()`.

TREE STRUCTURE

The root of the tree is dev_cfg, an array of usb_cfg_data_t configuration nodes, each representing one device configuration. The array index does not correspond to a configuration's value; use the bConfigurationValue field of the configuration descriptor within to find out the proper number for a given configuration.

The size of the array is returned in dev_n_cfg. The array itself is not NULL terminated.

When USB_PARSE_LVL_ALL is returned in dev_parse_level, index 0 pertains to the first valid configuration. This pertains to device configuration 1 as USB configuration 0 is not defined. When dev_parse_level returns USB_PARSE_LVL_CFG or USB_PARSE_LVL_IF, index 0 pertains to the device's one configuration recognized by the system. (Note that the configuration level is the only descriptor level in the tree where the index value does not correspond to the descriptor's value.)

Each usb_cfg_data_t configuration node contains a parsed usb configuration descriptor (usb_cfg_descr_t cfg_descr) a pointer to its string description (char *cfg_str) and string size (cfg_strsize), a pointer to an array of interface nodes (usb_if_data_t *cfg_if), and a pointer to an array of class/vendor (cv) descriptor nodes (usb_cvs_data_t *cfg_cvs). The interface node array size is kept in cfg_n_if, and the cv node array size is kept in cfg_n_cvs; neither array is NULL terminated. When USB_PARSE_LVL_IF is returned in dev_parse_level, the only interface (or alternate group) included in the tree is that which is recognized by the system for the current OS device node.

Each interface can present itself potentially in one of several alternate ways. An alternate tree node (usb_alt_if_data_t) represents an alternate representation. Each usb_if_data_t interface node points to an array of alternate nodes (usb_alt_if_data_t *if_alt) and contains the size of the array (if_n_alt).

Each interface alternate node holds an interface descriptor (usb_if_descr_t altif_descr), a pointer to its string description (char *altif_str), and has its own set of endpoints and bound cv descriptors. The pointer to the array of endpoints is usb_ep_data_t *altif_ep); the endpoint array size is altif_n_ep. The pointer to the array of cv descriptors is usb_cvs_data_t *altif_cvs; the cv descriptor array size is altif_n_cvs.

Each endpoint node holds an endpoint descriptor (usb_ep_descr_t ep_descr), a pointer to an array of cv descriptors for that endpoint (usb_cvs_data_t *ep_cvs), and the size of that array (ep_n_cvs). An endpoint descriptor may be passed to `usb_pipe_open(9F)` to establish a logical connection for data transfer.

Class and vendor descriptors (cv descriptors) are grouped with the configuration, interface or endpoint descriptors they immediately follow in the raw data returned by the device. Tree nodes representing such descriptors (usb_cvs_data_t) contain a pointer to the raw data (uchar_t *cvs_buf) and the size of the data (uint_t cvs_buf_len).

Configuration and interface alternate nodes return string descriptions. Note that all string descriptions returned have a maximum length of USB_MAXSTRINGLEN bytes and are in English ASCII.

Examples In the following example, a device's configuration data, including the following descriptor tree, is retrieved by `usb_get_dev_data(9F)` into usb_client_dev_data_t *reg_data:

```
config 1
     iface 0
         alt 0
             endpt 0
 config 2
     iface 0
     iface 1
         alt 0
             endpt 0
                 cv 0
         alt 1
             endpt 0
             endpt 1
                 cv 0
             endpt 2
         alt 2
             endpt 0
                 cv 0

 and suppose that the C/V data is of the following format:

 typedef struct cv_data {
     char char1;
     short short1;
     char char2;
 } cv_data_t;

 Parse the data of C/V descriptor 0, second configuration
 (index 1), iface 1, alt 2, endpt 0.

 usb_client_dev_data_t reg_data;
 usb_cvs_data_t *cv_node;
 cv_data_t parsed_data;

 cv_node =
     &reg_data->dev_cfg[1].cfg_if[1].if_alt[2].altif_ep[0].ep_cvs[0];
 (void)usb_parse_data("csc",
     (void *)(&cv_node->cvs_buf), cv_node->cvs_buf_len,
```

```
                    &parsed_data, sizeof(cv_data_t));
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb

See Also usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_lookup_ep_data(9F), usb_parse_data(9F), usb_pipe_open(9F), usb_cfg_descr(9S), usb_if_descr(9S), usb_ep_descr(9S), usb_string_descr(9S)

Name usb_completion_reason – USB completion reason definitions

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description If an error occurs during execution of a USB request, the USBA framework calls a client driver's exception callback handler to relay what happened. The host controller reports transport errors to the exception callback handler through the handler's request argument's completion reason (usb_cr_t) field. A completion reason of USB_CR_OK means the transfer completed with no errors detected.

The usb_cr_t enumerated type contains the following definitions:

USB_CR_OK
: The transfer completed without any errors being detected.

USB_CR_CRC
: CRC error was detected.

USB_CR_BITSTUFFING
: Bit stuffing violation was detected.

USB_CR_DATA_TOGGLE_MM
: Data toggle packet identifier did not match expected value.

USB_CR_STALL
: The device endpoint indicated that it is stalled. If autoclearing is enabled for the request (request attributes has USB_ATTRS_AUTOCLEARING set), check the callback flags (usb_cb_flags_t) in the callback handler to determine whether the stall is a functional stall (USB_CB_FUNCTIONAL_STALL) or a protocol stall (USB_CB_PROTOCOL_STALL). Please see usb_request_attributes(9S) for more information on autoclearing.

USB_CR_DEV_NOT_RESP
: Host controller timed out while waiting for device to respond.

USB_CR_PID_CHECKFAILURE
: Check bits on the packet identifier returned from the device were not as expected.

USB_CR_UNEXP_PID
: Packet identifier received was not valid.

USB_CR_DATA_OVERRUN
: Amount of data returned exceeded either the maximum packet size of the endpoint or the remaining buffer size.

USB_CR_DATA_UNDERRUN
: Amount of data returned was not sufficient to fill the specified buffer and the USB_ATTRS_SHORT_XFER_OK attribute was not

set. Please see usb_request_attributes(9S) for more information on allowance of short transfers.

USB_CR_BUFFER_OVERRUN — A device sent data faster than the system could digest it.

USB_CR_BUFFER_UNDERRUN — The host controller could not get data from the system fast enough to keep up with the required USB data rate.

USB_CR_TIMEOUT — A timeout specified in a control, bulk, or one-time interrupt request has expired.

USB_CR_NOT_ACCESSED — Request was not accessed nor processed by the host controller.

USB_CR_NO_RESOURCES — No resources were available to continue servicing a periodic interrupt or isochronous request.

USB_CR_STOPPED_POLLING — Servicing of the current periodic request cannot continue because polling on an interrupt-IN or isochronous-IN endpoint has stopped.

USB_CR_PIPE_CLOSING — Request was not started because the pipe to which it was queued was closing or closed.

USB_CR_PIPE_RESET — Request was not started because the pipe to which it was queued was reset.

USB_CR_NOT_SUPPORTED — Request or command is not supported.

USB_CR_FLUSHED — Request was not completed because the pipe to which it was queued went to an error state, became stalled, was reset or was closed.

USB_CR_HC_HARDWARE_ERR — Request could not be completed due to a general host controller hardware error.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb, SUNWusbu

See Also

usb_alloc_request(9F), usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_bulk_request(9S), usb_ctrl_request(9S), usb_intr_request(9S), usb_isoc_request(9S).

Name usb_ctrl_request – USB control pipe request structure

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description A control request is used to send device commands (or requests) and to read status. Please refer to Section *5.5* of the *USB 2.0* specification for information on control pipes. For information on formatting requests, see Section *9.3* of the *USB 2.0* specification. The USB 2.0 specification is available at *www.usb.org*.

Structure Members The fields in the usb_ctrl_req_t are used to format a control request:

```
uint8_t      ctrl_bmRequestType; /* characteristics of request */
uint8_t      ctrl_bRequest;  /* specific request             */
uint16_t     ctrl_wValue;    /* varies according to request  */
uint16_t     ctrl_wIndex;    /* index or offset              */
uint16_t     ctrl_wLength;   /* number of bytes to xfer      */
mblk_t       *ctrl_data;     /* data for the data phase  */
                             /* IN or OUT: allocated by client */
uint_t       ctrl_timeout;   /* time until  USBA framework */
                             /* retires req, in seconds */
                             /* If set to zero, defaults to 5 sec */
usb_opaque_t    ctrl_client_private; /* client private info */
usb_req_attrs_t ctrl_attributes; /* attrib. for this req */

/* Normal callback function, called upon completion. */
void       (*ctrl_cb)(
               usb_pipe_handle_t ph, struct usb_ctrl_req *req);

/* Exception callback function, for error handling. */
void       (*ctrl_exc_cb)(
               usb_pipe_handle_t ph, struct usb_ctrl_req *req);
usb_cr_t   ctrl_completion_reason; /* overall success status */
                                /* See usb_completion_reason(9S) */
usb_cb_flags_t  ctrl_cb_flags; /* recovery done by callback hndlr */
                              /* See usb_callback_flags(9S) */
```

Request attributes define special handling for transfers. The following attributes are valid for control requests:

USB_ATTRS_SHORT_XFER_OK — Accept transfers where less data is received than expected.

USB_ATTRS_AUTOCLEARING — Have USB framework reset pipe and clear functional stalls automatically on exception.

USB_ATTRS_PIPE_RESET — Have USB framework reset pipe automatically on exception.

Please see `usb_request_attributes(9S)` for more information.

The following definitions directly pertain to fields in the USB control request structure. (See Section *9.3* of the *USB 2.0* specification.)

```
Direction bitmasks of a control request's ctrl_bmRequestType field
(USB 2.0 spec, section 9.3.1)

     USB_DEV_REQ_HOST_TO_DEV     | Host to device direction
     USB_DEV_REQ_DEV_TO_HOST     | Device to host direction
     USB_DEV_REQ_DIR_MASK        | Bitmask of direction bits

Request type bitmasks of a control request's ctrl_bmRequestType field
(USB 2.0 spec, section 9.3.1)

     USB_DEV_REQ_TYPE_STANDARD   | USB 2.0 defined command
                                 | for all USB devices
     USB_DEV_REQ_TYPE_CLASS      | USB 2.0 defined
                                 | class-specific command
     USB_DEV_REQ_TYPE_VENDOR     | Vendor-specific command
     USB_DEV_REQ_TYPE_MASK       | Bitmask of request type bits

Recipient bitmasks of a control request's ctrl_bmRequestType field
(USB 2.0 spec, section 9.3.1)

     USB_DEV_REQ_RCPT_DEV        | Request is for device
     USB_DEV_REQ_RCPT_IF         | Request is for interface
     USB_DEV_REQ_RCPT_EP         | Request is for endpoint
     USB_DEV_REQ_RCPT_OTHER      | Req is for other than above
     USB_DEV_REQ_RCPT_MASK       | Bitmask of request recipient bits

Standard requests (USB 2.0 spec, section 9.4)
     USB_REQ_GET_STATUS          | Get status of device, endpoint
                                 |or interface (9.4.5)
     USB_REQ_CLEAR_FEATURE       | Clear feature specified by
                                 |wValue field (9.4.1)
     USB_REQ_SET_FEATURE         | Set feature specified by
                                 |      wValue field (9.4.9)
     USB_REQ_SET_ADDRESS         | Set address specified by
                                 |      wValue field (9.4.6)
     USB_REQ_GET_DESCR           | Get descr for item/idx in
                                 |      wValue field (9.4.3)
     USB_REQ_SET_DESCR           | Set descr for item/idx in
                                 |      wValue field (9.4.8)
     USB_REQ_GET_CFG             | Get current device
                                 |       configuration (9.4.2)
     USB_REQ_SET_CFG             | Set current device
                                 |       configuration (9.4.7)
     USB_REQ_GET_IF              | Get alternate interface
                                 |      setting (9.4.4)
```

```
     USB_REQ_SET_IF              | Set alternate interface
                                 |       setting (9.4.10)
     USB_REQ_SYNC_FRAME          | Set and report an endpoint's
                                 |       sync frame (9.4.11)

Unicode language ID, used as wIndex for USB_REQ_SET/GET_DESCRIPTOR

     USB_LANG_ID                 | Unicode English Lang ID for
                                 | parsing str descr
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also usb_alloc_request(9F), usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_bulk_request(9S), usb_callback_flags(9S), usb_completion_reason(9S), usb_intr_request(9S), usb_isoc_request(9S), usb_request_attributes(9S)

Name usb_dev_descr – USB device descriptor

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description The `usb_dev_descr_t` device descriptor defines device-wide attributes. Please refer to Section *9.6.1* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

The device descriptor is retrieved from a USB device during device enumeration and can be accessed via `usb_get_dev_data(9F)`.

A device descriptor contains the following fields:

```
uint8_t     bLength             Size of this descriptor,
                                in bytes.

uint8_t     bDescriptorType     Set to USB_DESCR_TYPE_DEV.

uint16_t    bcdUSB              USB specification release
                                number supported, in bcd.

uint8_t     bDeviceClass        Class code (see below).

uint8_t     bDeviceSubClass     Subclass code (see USB 2.0
                                specification of applicable
                                device class for information.)

uint8_t     bDeviceProtocol     Protocol code (see USB 2.0
                                specification of applicable
                                device class for information.)

uint8_t     bMaxPacketSize0     Maximum packet size of
                                endpoint 0.

uint16_t    idVendor            vendor ID value.

uint16_t    idProduct           product ID value.

uint16_t    bcdDevice           Device release number in
                                binary coded decimal.

uint8_t     iManufacturer       Index of optional manufacturer
                                description string.
                                Valid if > 0.

uint8_t     iProduct            Index of optional product
                                description string.
                                Valid if > 0.
```

```
uint8_t     iSerialNumber       Index of optional serial
                                number string.
                                Valid if > 0.

uint8_t     bNumConfigurations  Number of available
                                configurations.


Device descriptors bDeviceClass values:

USB_CLASS_PER_INTERFACE         Class information is at
                                interface level.

USB_CLASS_COMM                  CDC control device class.

USB_CLASS_DIAG                  Diagnostic device class.

USB_CLASS_HUB                   HUB device class.

USB_CLASS_MISC                  MISC device class.

USB_CLASS_VENDOR_SPEC           Vendor-specific class.

USB_CLASS_WIRELESS              Wireless controller
                                device class.
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also

attributes(5), usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_parse_data(9F), usb_cfg_descr(9S), usb_ctrl_request(9S), usb_dev_qlf_descr(9S), usb_ep_descr(9S), usb_if_descr(9S), usb_other_speed_cfg_descr(9S), usb_string_descr(9S)

Name usb_dev_qlf_descr – USB device qualifier descriptor

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The device qualifier descriptor `usb_dev_qlf_descr_t` defines how fields of a high speed device's device descriptor would look if that device is run at a different speed. If a high-speed device is running currently at full/high speed, fields of this descriptor reflect how device descriptor fields would look if speed was changed to high/full. Please refer to section *9.6.2* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

A device descriptor contains the following fields:

```
uint8_t      bLength              Size of this descriptor.

uint8_t      bDescriptorType      Set to USB_DESCR_TYPE_DEV_QLF.

uint16_t     bcdUSB               USB specification release
                                  number in binary coded decimal.

uint8_t      bDeviceClass         Device class code.
                                  (See usb_dev_descr(9s).)

uint8_t      bDeviceSubClass      Device subclass code.(See
                                  USB 2.0 specification of
                                  applicable device class for
                                  information.)

uint8_t      bDeviceProtocol      Protocol code.(See
                                  USB 2.0 specification of
                                  applicable device class for
                                  information.)

uint8_t      bMaxPacketSize0      Maximum packet size of
                                  endpoint 0.

uint8_t      bNumConfigurations   Number of available
                                  configurations.

uint8_t      bReserved            Reserved.
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWusbu

See Also attributes(5), usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F). usb_parse_data(9F), usb_ctrl_request(9S), usb_cfg_descr(9S), usb_dev_descr(9S), usb_ep_descr(9S), usb_if_descr(9S), usb_other_speed_cfg_descr(9S), usb_string_descr(9S)

Name usb_ep_descr – USB endpoint descriptor

Synopsis
```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `usb_ep_descr_t` endpoint descriptor defines endpoint attributes. An endpoint is a uniquely addressable portion of a USB device that is a source or sink of data.

Please refer to Section *9.6.6* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

One or more endpoint descriptors are retrieved from a USB device during device enumeration. They can be accessed via `usb_get_dev_data`(9F).

A endpoint descriptor has the following fields:

```
uint8_t     bLength                 Size of this descriptor
                                    in bytes.

uint8_t     bDescriptorType         Set to USB_DESCR_TYPE_EP.

uint8_t     bEndpointAddress        Endpoint address.

uint8_t     bmAttributes            Endpoint attrib. (see below.)

uint16_t    wMaxPacketSize          Maximum pkt size.

uint8_t     bInterval               Polling interval for interrupt
                                    and isochro. endpoints.
                                    NAK rate for high-speed control
                                    and bulk endpoints.

Endpoint descriptor bEndpointAddress bitmasks contain address number
and direction fields as follows:
    USB_EP_NUM_MASK        Address bits
    USB_EP_DIR_MASK        Direction bit
    USB_EP_DIR_OUT         OUT towards device
    USB_EP_DIR_IN          IN towards host

Endpoint descriptor transfer type bmAttributes values and mask:
    USB_EP_ATTR_CONTROL    Endpoint supports control transfers
    USB_EP_ATTR_ISOCH      Endpoint supports isochronous xfers
    USB_EP_ATTR_BULK       Endpoint supports bulk transfers
    USB_EP_ATTR_INTR       Endpoint supports interrupt transfers
    USB_EP_ATTR_MASK       bmAttributes transfer-type bit field

Endpoint descriptor synchronization type bmAttributes values and mask
for isochronous endpoints:
```

```
    USB_EP_SYNC_NONE       Endpoint supports no synchronization
    USB_EP_SYNC_ASYNC      Endpoint supports asynchronous sync
    USB_EP_SYNC_ADPT       Endpoint supports adaptive sync
    USB_EP_SYNC_SYNC       Endpoint supports synchronous sync
    USB_EP_SYNC_MASK       bmAttributes sync type bit field

Endpoint descriptor feedback type bmAttributes values and mask for
isochronous endpoints:
    USB_EP_USAGE_DATA                 Data endpoint
    USB_EP_USAGE_FEED                 Feedback endpoint
    USB_EP_USAGE_IMPL                 Implicit feedback data endpoint
    USB_EP_USAGE_MASK                 bmAttributes feedback type bit fld

Endpoint descriptor additional-transaction-opportunities-
per-microframe wMaxPacketSize values and mask for high speed
isochronous and interrupt endpoints:
    USB_EP_MAX_PKTSZ_MASK             Mask for packetsize bits
    USB_EP_MAX_XACTS_MASK             Bits for additional transfers per
                                      microframe
    USB_EP_MAX_XACTS_SHIFT            Left-shift this number of bits to
                                      get to additional-transfers-per-
                                      microframe bitfield

Endpoint descriptor polling bInterval range values:
    USB_EP_MIN_HIGH_CONTROL_INTRVL    Min NAK rate for highspd ctrl e/p
    USB_EP_MAX_HIGH_CONTROL_INTRVL    Max NAK rate for highspd ctrl e/p

    USB_EP_MIN_HIGH_BULK_INTRVL       Min NAK rate for highspd bulk e/p
    USB_EP_MAX_HIGH_BULK_INTRVL       Max NAK rate for highspd bulk e/p

    USB_EP_MIN_LOW_INTR_INTRVL        Min poll interval, lowspd intr e/p
    USB_EP_MAX_LOW_INTR_INTRVL        Max poll interval, lowspd intr e/p

    USB_EP_MIN_FULL_INTR_INTRVL       Min poll interval, fullspd intr e/p
    USB_EP_MAX_FULL_INTR_INTRVL       Max poll interval, fullspd intr e/p

Note that for the following polling bInterval range values, the interval
is 2**(value-1). See Section 9.6.6 of the USB 2.0 specification.

    USB_EP_MIN_HIGH_INTR_INTRVL       Min poll interval, highspd intr e/p
    USB_EP_MAX_HIGH_INTR_INTRVL       Max poll interval, highspd intr e/p

    USB_EP_MIN_FULL_ISOCH_INTRVL      Min poll interval, fullspd isoc e/p
    USB_EP_MAX_FULL_ISOCH_INTRVL      Max poll interval, fullspd isoc e/p

    USB_EP_MIN_HIGH_ISOCH_INTRVL      Min poll interval, highspd isoc e/p
    USB_EP_MAX_HIGH_ISOCH_INTRVL      Max poll interval, highspd isoc e/p
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also attributes(5), usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_parse_data(9F), usb_cfg_descr(9S), usb_ctrl_request(9S), usb_dev_descr(9S), usb_dev_qlf_descr(9S), usb_if_descr(9S), usb_other_speed_cfg_descr(9S), usb_string_descr(9S)

Name usb_if_descr – USB interface descriptor

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description The `usb_if_descr_t` interface descriptor defines attributes of an interface. A configuration contains one or more interfaces. An interface contains one or more endpoints.

Please refer to Section *9.6.5* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

One or more configuration descriptors are retrieved from a USB device during device enumeration. They can be accessed via `usb_get_dev_data(9F)`.

A interface descriptor has the following fields:

```
uint8_t     bLength                 Size of this descriptor
                                    in bytes.

uint8_t     bDescriptorType         Set to USB_DESCR_TYPE_IF.

uint8_t     bInterfaceNumber        Interface number (0-based).

uint8_t     bAlternateSetting       Alternate setting number for
                                    this interface and its
                                    endpoints (0-based).

uint8_t     bNumEndpoints           Number of endpoints,
                                    excluding endpoint 0.

uint8_t     bInterfaceClass         Interface Class code
                                    (see below).

uint8_t     bInterfaceSubClass      Sub class code. (See USB 2.0
                                    specification of applicable
                                    interface class for information.)

uint8_t     bInterfaceProtocol      Protocol code.  (See USB 2.0
                                    specification of applicable
                                    interface class for information.)

uint8_t     iInterface              Index of optional string
                                    describing this interface
                                    Valid if > 0. Pass to
                                    usb_get_string_descr(9F) to
                                    retrieve string.
```

```
USB 2.0 specification interface descriptor bInterfaceClass field
```

```
values are as follows:

        USB_CLASS_APP           Application-specific interface class

        USB_CLASS_AUDIO         Audio interface class

        USB_CLASS_CCID          Chip/Smartcard interface class

        USB_CLASS_CDC_CTRL      CDC control interface class

        USB_CLASS_CDC_DATA      CDC data interface class

        USB_CLASS_SECURITY      Content security interface class

        USB_CLASS_DIAG          Diagnostic interface class

        USB_CLASS_HID           HID interface class

        USB_CLASS_HUB           HUB interface class

        USB_CLASS_MASS_STORAGE  Mass storage interface class

        USB_CLASS_PHYSICAL      Physical interface class

        USB_CLASS_PRINTER       Printer interface class

        USB_CLASS_VENDOR_SPEC   Vendor-specific interface class

        USB_CLASS_WIRELESS      Wireless interface class
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also

attributes(5), usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_parse_data(9F), usb_cfg_descr(9S), usb_ctrl_request(9S), usb_dev_descr(9S), usb_dev_qlf_descr(9S), usb_ep_descr(9S), usb_other_speed_cfg_descr(9S), usb_string_descr(9S)

Name usb_intr_request – USB interrupt request structure

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description An interrupt request (that is, a request sent through an interrupt pipe), is used to transfer small amounts of data infrequently, but with bounded service periods. (Data flows in either direction.) Please refer to Section *5.7* of the *USB 2.0* specification for information on interrupt transfers. (The *USB 2.0* specification is available at *www.usb.org*.)

The fields in the usb_intr_req_t are used to format an interrupt request. Please see below for acceptable combinations of flags and attributes.

The usb_intr_req_t fields are:

```
ushort_t       intr_len;     /* Size of pkt. Must be set */
                             /* Max size is 8K for low/full speed */
                             /* Max size is 20K for high speed */
mblk_t         *intr_data;   /* Data for the data phase  */
                             /* IN: zero-len mblk alloc by client */
                             /* OUT: allocated by client */
usb_opaque_t   intr_client_private; /* client specific information  */
uint_t         intr_timeout; /* only with ONE TIME POLL, in secs */
                             /* If set to zero, defaults to 5 sec */
usb_req_attrs_t intr_attributes;

/* Normal callback function, called upon completion. */
void           (*intr_cb)(
                   usb_pipe_handle_t ph, struct usb_intr_req *req);

/* Exception callback function, for error handling. */
void           (*intr_exc_cb)(
                   usb_pipe_handle_t ph, struct usb_intr_req *req);

/* set by USBA/HCD on completion */
usb_cr_t    intr_completion_reason; /* overall completion status */
                                   /* See usb_completion_reason(9S) */
usb_cb_flags_t intr_cb_flags; /* recovery done by callback hndlr */
                                  /* See usb_callback_flags(9S) */
```

Request attributes define special handling for transfers. The following attributes are valid for interrupt requests:

USB_ATTRS_SHORT_XFER_OK — Accept transfers where less data is received than expected.

USB_ATTRS_AUTOCLEARING — Have USB framework reset pipe and clear functional stalls automatically on exception.

USB_ATTRS_PIPE_RESET	Have USB framework reset pipe automatically on exception.
USB_ATTRS_ONE_XFER	Perform a single IN transfer. Do not start periodic transfers with this request.

Please see usb_request_attributes(9S) for more information.

```
Interrupt transfers/requests are subject to the following
constraints and caveats:

1) The following table indicates combinations of
   usb_pipe_intr_xfer() flags argument and fields
   of the usb_intr_req_t request argument (X = don't care):

     "none" as attributes in the table below indicates
     neither ONE_XFER nor SHORT_XFER_OK

     flags     Type  attributes      data    timeout semantics
     ----------------------------------------------------------------
     X         IN    X               !=NULL  X       illegal

     X         IN    !ONE_XFER       X       !=0     illegal

     X         IN    !ONE_XFER       NULL    0       See table note (A)

     no sleep  IN    ONE_XFER        NULL    0       See table note (B)

     no sleep  IN    ONE_XFER        NULL    !=0     See table note (C)

     sleep     IN    ONE_XFER        NULL    0       See table note (D)

     sleep     IN    ONE_XFER        NULL    !=0     See table note (E)

     X         OUT   X               NULL    X       illegal

     X         OUT   ONE_XFER        X       X       illegal

     X         OUT   SHORT_XFER_OK   X       X       illegal

     no sleep  OUT   none            !=NULL  0       See table note (F)

     no sleep  OUT   none            !=NULL  !=0     See table note (G)

     sleep     OUT   none            !=NULL  0       See table note (H)

     sleep     OUT   none            !=NULL  !=0     See table note (I)
```

```
Table notes:

  A) Continuous polling, new data is returned in cloned request
     structures via continous callbacks, original request is
     returned on stop polling.

  B) One time poll, no timeout, callback when data is
     received.

  C) One time poll, with timeout, callback when data
     is received.

  D) One time poll, no timeout, one callback, unblock
     when transfer completes.

  E) One time poll, timeout, one callback, unblock when
     transfer completes or timeout occurs.

  F) Transfer until data exhausted, no timeout, callback
     when done.

  G) Transfer until data exhausted, timeout, callback
     when done.

  H) Transfer until data exhausted, no timeout, unblock
     when data is received.

  I) Transfer until data exhausted, timeout, unblock
     when data is received.


2) USB_FLAGS_SLEEP indicates here just to wait for
   resources, except when ONE_XFER is set, in which case it
   also waits for completion before returning.

3) Reads (IN):

  a) The client driver does *not* provide a data buffer.
     By default, a READ request would mean continuous
     polling for data IN.  The USBA framework allocates a
     new data buffer for each poll.  intr_len specifies
    the amount of 'periodic data' for each poll.

  b) The USBA framework issues a callback to the client
     at the end of a polling interval when there is data to
     return.  Each callback returns its data in a new request
     cloned from the original.  Note that the amount of data
```

```
   read IN is either intr_len or "wMaxPacketSize" in length.

c) Normally, the HCD keeps polling the interrupt endpoint
   forever even if there is no data to be read IN.  A
   client driver may stop this polling by calling
   usb_pipe_stop_intr_polling(9F).

d) If a client driver chooses to pass USB_ATTRS_ONE_XFER
   as 'xfer_attributes' the HCD polls for data until
   some data is received.  The USBA framework reads in
   the data, does a callback, and stops polling for any
   more data.  In this case, the client
   driver need not explicitly call
   usb_pipe_stop_intr_polling().

e) All requests with USB_ATTRS_ONE_XFER require callbacks
   to be specified.

f) When continuous polling is stopped, the original
   request is returned with USB_CR_STOPPED_POLLING.

g) If the USB_ATTRS_SHORT_XFER_OK attribute is not set
   and a short transfer is received while polling,
   an error is assumed and polling is stopped.  In this
   case or the case of other errors, the error must be cleared
   and polling restarted by the client driver. Setting the
   USB_ATTRS_AUTOCLEARING attribute will clear the error
   but not restart polling.  (NOTE: Polling can be
   restarted from an exception callback corresponding to
   an original request.  Please see usb_pipe_intr_xfer(9F)
   for more information.

4) Writes (OUT):

a) A client driver provides the data buffer, and
   data, needed for intr write.

b) Unlike read (see previous section), there
  is no continuous write mode.

c) The USB_ATTRS_ONE_XFER attribute is illegal.
   By default USBA keeps writing intr data until
   the provided data buffer has been
   written out. The USBA framework does ONE
   callback to the client driver.

d) Queueing is supported.
```

```
The intr_completion_reason indicates the status
of the transfer.  See usb_completion_reason(9S) for
usb_cr_t definitions.

The intr_cb_flags are set prior to calling the
exception callback handler, to summarize recovery actions
taken and errors encountered during
recovery.  See usb_callback_flags(9S) for
usb_cb_flags_t definitions.

--- Callback handling ---

All usb request types share the same callback
handling. Please see usb_callback_flags(9S) for a
description of use and operation.
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also usb_alloc_request(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_bulk_xfer(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_bulk_request(9S), usb_callback_flags(9S), usb_completion_reason(9S), usb_ctrl_request(9S), usb_isoc_request(9S), usb_request_attributes(9S)

Name usb_isoc_request – USB isochronous request structure

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description A request sent through an isochronous pipe is used to transfer large amounts of packetized data with relative unreliability, but with bounded service periods. A packet is guaranteed to be tried within a bounded time period, but is not retried upon failure. Isochronous transfers are supported on both USB 1.1 and USB 2.0 devices. For further information, see section *5.6* of the *USB 2.0* specification available at *www.usb.org*.

This section provides information on acceptable combinations of flags and attributes with additional details. The following fields of the usb_isoc_req_t are used to format an isochronous request.

```
usb_frame_number_t
                isoc_frame_no;   /* frame num to start sending req. */
ushort_t       isoc_pkts_count; /* num USB pkts in this request */
/*
 * The sum of all pkt lengths in an isoc request. Recommend to set it to
 * zero, so the sum of isoc_pkt_length in the isoc_pkt_descr list will be
 * used automatically and no check will be apply to this element.
 */
ushort_t       isoc_pkts_length;
ushort_t       isoc_error_count;/* num pkts completed w/errs */
usb_req_attrs_t isoc_attributes;/* request-specific attrs */
mblk_t          *isoc_data;     /* data to xfer */
                                 /* IN or OUT: alloc. by client. */
                                 /* Size=total of all pkt lengths. */
usb_opaque_t     isoc_client_private; /* for client driver excl use. */
struct usb_isoc_pkt_descr       /* (see below) */
                 *isoc_pkt_descr;

/*
 * Normal callback function, called upon completion.
 * This function cannot block as it executes in soft interrupt context.
 */
void       (*isoc_cb)(
                usb_pipe_handle_t ph, struct usb_isoc_req *req);

/* Exception callback function, for error handling. */
void       (*isoc_exc_cb)(
               usb_pipe_handle_t ph, struct usb_isoc_req *req);

usb_cr_t  isoc_completion_reason; /* overall completion status */
                                   /* set by USBA framework */
                                   /* See usb_completion_reason(9S) */
usb_cb_flags_t  isoc_cb_flags;    /* recovery done by callback hndlr */
```

```
                                            /* set by USBA on exception. */
                                            /* See usb_callback_flags(9S) */
```

A `usb_isoc_pkt_descr_t` describes the status of an isochronous packet transferred within a frame or microframe. The following fields of a `usb_isoc_pkt_descr_t` packet descriptor are used within an `usb_isoc_req_t`. The `isoc_pkt_length` is set by the client driver to the amount of data managed by the packet for input or output. The latter two fields are set by the USBA framework to indicate status. Any packets with an `isoc_completion_reason`, other than `USB_CR_OK`, are reflected in the `isoc_error_count` of the `usb_isoc_req_t`.

```
ushort_t    isoc_pkt_length;        /* number bytes to transfer */
 ushort_t    isoc_pkt_actual_length; /* actual number transferred */
 usb_cr_t    isoc_pkt_status;        /* completion status */
```

If two multi-frame `isoc` requests that both specify the `USB_ATTRS_ISOC_XFER_ASAP` attribute are scheduled closely together, the first frame of the second request is queued to start after the last frame of the first request.

No stalls are seen in isochronous transfer exception callbacks. Because transfers are not retried upon failure, isochronous transfers continue regardless of errors.

Request attributes define special handling for transfers. The following attributes are valid for isochronous requests:

`USB_ATTRS_ISOC_START_FRAME` — Start transferring at the starting frame number specified in the `isoc_frame_no` field of the request.

`USB_ATTRS_ISOC_XFER_ASAP` — Start transferring as soon as possible. The USBA framework picks an immediate frame number to map to the starting frame number.

`USB_ATTRS_SHORT_XFER_OK` — Accept transfers where less data is received than expected.

The `usb_isoc_req_t` contains an array of descriptors that describe isochronous packets. One isochronous packet is sent per frame or microframe. Because packets that comprise a transfer are sent across consecutive frames or microframes, `USB_ATTRS_ONE_XFER` is invalid.

See `usb_request_attributes(9S)` for more information.

Isochronous transfers/requests are subject to the following constraints and caveats:

```
1) The following table indicates combinations of usb_pipe_isoc_xfer
   flags argument and fields of the usb_isoc_req_t request argument
   (X = don't care). (Note that attributes considered in this table
   are ONE_XFER, START_FRAME, XFER_ASAP, and SHORT_XFER, and that
   some transfer types are characterized by multiple table entries.)

Flags Type     Attributes          Data    Semantics
---------------------------------------------------------------
X      X      X                    NULL    illegal
```

```
X      X      ONE_XFER             X       illegal

X      X      ISOC_START_FRAME     X       illegal
              & ISOC_XFER_ASAP

X      X      !ISOC_START_FRAME    X       illegal
              & !ISOC_XFER_ASAP

X      OUT    SHORT_XFER_OK        X       illegal

X      IN     X                    !=NULL  See table note (A)

X      X      ISOC_START_FRAME     !=NULL  See table note (B)

X      X      ISOC_XFER_ASAP       !=NULL  See table note (C)

Table notes:

    A) continuous polling, new data is returned in
       cloned request structures via continous callbacks,
       original request is returned on stop polling

    B) invalid if the current_frame number is past
       "isoc_frame_no" or "isoc_frame_no" == 0

    C)"isoc_frame_no" is ignored. The USBA framework
       determines which frame to insert and start
       the transfer.

2) USB_FLAGS_SLEEP indicates to wait for resources but
   not for completion.

3) For polled reads:

  A. The USBA  framework  accepts  a  request  which
     specifies  the  size and number of packets to fill
     with data. The packets get filled one  packet  per
     (1  ms)  frame/(125 us) microframe.  All  requests
     have an implicit USB_ATTRS_SHORT_XFER_OK attribute
     set, since transfers  continue in spite of any en-
     countered. The amount of data read per packet  will
     match  the  isoc_pkt_length  field  of  the packet
     descriptor unless a  short  transfer  occurs.  The
     actual     size     is     returned     in     the
     isoc_pkt_actual_length   field   of   the   packet
     descriptor.  When  all packets of the request have
```

```
      been processed, a normal callback is done to  sig-
      nal the completion of the original request.

   B. When continuous polling is stopped, the original
      request is returned in an exception callback with a
      completion reason of USB_CR_STOPPED_POLLING.
      (NOTE: Polling can be restarted from  an exception
      callback corresponding to an original request.
      Please see usb_pipe_isoc_xfer(9F) for more information.

   C. Callbacks must be specified.

   The isoc_completion_reason indicates the status of the transfer. See
   usb_completion_reason(9s) for usb_cr_t definitions.

   The isoc_cb_flags are set prior to calling the exception
   callback handler to summarize recovery actions taken and
   errors encountered during recovery. See usb_callback_flags(9s)
   for usb_cb_flags_t definitions.

--- Callback handling ---
All usb request types share the same callback handling. Please see
usb_callback_flags(9s) for a description of use and operation.
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also

attributes(5), usb_alloc_request(9F), usb_get_current_frame_number(9F), usb_get_max_pkts_per_isoc_request(9F), usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_bulk_request(9S), usb_callback_flags(9S), usb_completion_reason(9S), usb_ctrl_request(9S), usb_intr_request(9S), usb_request_attributes(9S)

Name usb_other_speed_cfg_descr – USB other speed configuration descriptor

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description The `usb_other_speed_cfg_descr_t` configuration descriptor defines how fields of a high speed device's configuration descriptor change if that device is run at its other speed. Fields of this descriptor reflect configuration descriptor field changes if a device's speed is changed from full to high speed, or from high to full speed.

Please refer to Section *9.6.4* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

This descriptor has the following fields:

```
uint8_t     bLength                Size of this descriptor,
                                   in bytes.

uint8_t     bDescriptorType        Set to USB_DESCR_TYPE_OTHER_SPEED_CFG.

uint16_t    wTotalLength           Total length of data returned */
                                   including all descriptors in
                                   the current other-speed
                                   configuration.

uint8_t     bNumInterfaces         Number of interfaces in
                                   the selected configuration.

uint8_t     bConfigurationValue    ID of the current other-speed
                                   configuration (1-based).

uint8_t     iConfiguration         Configuration value.
                                   Valid if > 0. Pass to
                                   usb_get_string_descr(9F)
                                   to retrieve string.

uint8_t     bmAttributes           Configuration characteristics
                                   [See usb_cfg_descr(9S).]

uint8_t     bMaxPower              Maximum power consumption
                                   in 2mA units.
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface stability	Evolving
Availability	SUNWusbu

See Also attributes(5), usb_get_alt_if(9F), usb_get_cfg(9F), usb_get_dev_data(9F), usb_get_string_descr(9F), usb_parse_data(9F). usb_cfg_descr(9S), usb_ctrl_request(9S), usb_dev_descr(9S), usb_dev_qlf_descr(9S)

Name usb_request_attributes – Definition of USB request attributes

Synopsis

```
#include <sys/usb/usba.h>
```

Interface Level Solaris DDI specific (Solaris DDI)

Description Request attributes specify how the USBA framework handles request execution. Request attributes are specified in the request's *_attributes field and belong to the enumerated type usb_req_attrs_t.

Supported request attributes are:

USB_ATTRS_SHORT_XFER_OK
: Use this attribute when the maximum transfer size is known, but it is possible for the request to receive a smaller amount of data. This attribute tells the USBA framework to accept without error transfers which are shorter than expected.

USB_ATTRS_PIPE_RESET
: Have the USB framework reset the pipe automatically if an error occurs during the transfer. Do not attempt to clear any stall. The USB_CB_RESET_PIPE callback flag is passed to the client driver's exception handler to show the pipe has been reset. Pending requests on pipes which are reset are flushed unless the pipe is the default pipe.

USB_ATTRS_AUTOCLEARING
: Have the USB framework reset the pipe and clear functional stalls automatically if an error occurs during the transfer. The callback flags passed to the client driver's exception handler show the status after the attempt to clear the stall.

  USB_CB_FUNCTIONAL_STALL is set in the callback flags to indicate that a functional stall occurred. USB_CB_STALL_CLEARED is also set if the stall is cleared. The default pipe never shows a functional stall if the USB_ATTRS_AUTOCLEARING attribute is set. If USB_CB_FUNCTIONAL_STALL is seen when autoclearing is enabled, the device has a fatal error.

  USB_CB_PROTOCOL_STALL is set without USB_CB_STALL_CLEARED in the callback flags to indicate that a protocol stall was seen but was not explicitly cleared. Protocol stalls are cleared automatically when a subsequent command is issued.

Autoclearing a stalled default pipe is not allowed. The USB_CB_PROTOCOL_STALL callback flag is set in the callback flags to indicate the default pipe is stalled.

Autoclearing is not allowed when the request is USB_REQ_GET_STATUS on the default pipe.

USB_ATTRS_ONE_XFER
: Applies only to interrupt-IN requests. Without this flag, interrupt-IN requests start periodic polling of the interrupt pipe. This flag specifies to perform only a single transfer. Do not start periodic transfers with this request.

USB_ATTRS_ISOC_START_FRAME
: Applies only to isochronous requests and specifies that a request be started at a given frame number. The starting frame number is provided in the isoc_frame_no field of the usb_isoc_req_t. Please see `usb_isoc_request(9S)` for more information about isochronous requests.

  USB_ATTRS_ISOC_START_FRAME can be used to delay a transfer by a few frames, allowing transfers to an endpoint to sync up with another source. (For example, synching up audio endpoints to a video source.) The number of a suitable starting frame in the near future can be found by adding an offset number of frames (usually between four and ten) to the current frame number returned from `usb_get_current_frame_number(9F)`. Note that requests with starting frames which have passed are rejected.

USB_ATTRS_ISOC_XFER_ASAP
: Applies only to isochronous requests and specifies that a request start as soon as possible. The host controller driver picks a starting frame number which immediately follows the last frame of the last queued request. The isoc_frame_no of the usb_isoc_req_t is ignored. Please see `usb_isoc_request(9S)` for more information about isochronous requests.

Examples

```
/*
 * Allocate, initialize and issue a synchronous bulk-IN request.
 * Allow for short transfers.
 */

struct buf *bp;
usb_bulk_req_t bulk_req;
mblk_t *mblk;

bulk_req = usb_alloc_bulk_req(dip, bp->b_bcount, USB_FLAGS_SLEEP);

bulk_req->bulk_attributes =
    USB_ATTRS_AUTOCLEARING | USB_ATTRS_SHORT_XFER_OK;

if ((rval = usb_pipe_bulk_xfer(pipe, bulk_req, USB_FLAGS_SLEEP)) !=
    USB_SUCCESS) {
        cmn_err (CE_WARN, "%s%d: Error reading bulk data.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

mblk = bulk_req->bulk_data;
bcopy(mblk->rptr, buf->b_un.b_addr, mblk->wptr - mblk->rptr);
bp->b_resid = bp->b_count - (mblk->wptr = mblk->rptr);
...
...

----

usb_pipe_handle_t handle;
usb_frame_number_t offset = 10;
usb_isoc_req_t *isoc_req;

isoc_req = usb_alloc_isoc_req(...);
  ...
  ...
isoc_req->isoc_frame_no = usb_get_current_frame_number(dip) + offset;
isoc_req->isoc_attributes = USB_ATTRS_ISOC_START_FRAME;
  ...
  ...
if (usb_pipe_isoc_xfer(handle, isoc_req, 0) != USB_SUCCESS) {
  ...
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusb, SUNWusbu

See Also

usb_alloc_request(9F), usb_get_current_frame_number(9F), usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_intr_xfer(9F), usb_pipe_isoc_xfer(9F), usb_bulk_request(9S), usb_callback_flags(9S), usb_ctrl_request(9S), usb_intr_request(9S), usb_isoc_request(9S), usb_completion_reason(9S)

Name usb_string_descr – USB string descriptor

Synopsis `#include <sys/usb/usba.h>`

Interface Level Solaris DDI specific (Solaris DDI)

Description The `usb_string_descr_t` string descriptor defines the attributes of a string, including size and Unicode language ID. Other USB descriptors may have string descriptor index fields which refer to specific string descriptors retrieved as part of a device's configuration.

Please refer to Section *9.6.7* of the *USB 2.0* specification. The *USB 2.0* specification is available at *www.usb.org*.

A string descriptor has the following fields:

```
uint8_t       bLength                Size of this descriptor,
                                     in bytes.


uint8_t       bDescriptorType        Set to USB_DESCR_TYPE_STRING.

uint16_t      bString[1];            Variable length Unicode encoded
                                     string.
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface stability	Evolving
Availability	SUNWusbu

See Also `attributes(5)`, `usb_get_alt_if(9F)`, `usb_get_cfg(9F)`, `usb_get_dev_data(9F)`, `usb_get_string_descr(9F)`, `usb_parse_data(9F)`, `usb_ctrl_request(9S)`