



man pages section 9S: DDI and DKI Data Structures

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817-5426-10
February 2004

Copyright 2004 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2004 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, docs.sun.com, AnswerBook, AnswerBook2, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REpondre A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



031212@7518



Contents

Preface 5

Introduction 11

Intro(9S) 12

Data Structures for Drivers 15

aio_req(9S) 16

buf(9S) 17

cb_ops(9S) 20

copyreq(9S) 22

copyresp(9S) 23

datab(9S) 24

ddi_device_acc_attr(9S) 25

ddi_dma_attr(9S) 29

ddi_dma_cookie(9S) 32

ddi_dmae_req(9S) 33

ddi_dma_lim_sparc(9S) 37

ddi_dma_lim_x86(9S) 39

ddi_dma_req(9S) 41

ddi_idevice_cookie(9S) 44

ddi_mapdev_ctl(9S) 45

devmap_callback_ctl(9S) 46

dev_ops(9S) 48

fmodsw(9S) 49

free_rtn(9S) 50

iocblk(9S)	51
iovec(9S)	52
kstat(9S)	53
kstat_intr(9S)	55
kstat_io(9S)	56
kstat_named(9S)	57
linkblk(9S)	58
modldrv(9S)	59
modlinkage(9S)	60
modlstrmod(9S)	61
module_info(9S)	62
msgb(9S)	63
qband(9S)	64
qinit(9S)	65
queclass(9S)	66
queue(9S)	67
scsi_address(9S)	68
scsi_arq_status(9S)	69
scsi_asc_key_strings(9S)	70
scsi_device(9S)	71
scsi_extended_sense(9S)	72
scsi_hba_tran(9S)	75
scsi_inquiry(9S)	77
scsi_pkt(9S)	80
scsi_status(9S)	84
streamtab(9S)	86
stroptions(9S)	87
tuple(9S)	89
uio(9S)	92

Index 95

Preface

Both novice users and those familiar 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 <code>ioctl(2)</code> system call is called <code>ioctl</code> and generates its own heading. <code>ioctl</code> calls for a specific device are listed alphabetically (on the man page for that specific device). <code>ioctl</code> calls are used for a particular class of devices all of which have an <code>io</code> ending, such as <code>mtio(7I)</code> .
OPTIONS	This section 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 <code>errno</code> 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 <code>example%</code> , or if the user must be superuser, <code>example#</code> . 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 <code>attributes(5)</code> 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.

Introduction

Intro(9S)

NAME Intro – introduction to kernel data structures

DESCRIPTION Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

In this section, reference pages contain the following headings:

- **NAME** summarizes the structure’s purpose.
- **SYNOPSIS** lists the include file that defines the structure.
- **INTERFACE LEVEL** describes any architecture dependencies.
- **DESCRIPTION** provides general information about the structure.
- **STRUCTURE MEMBERS** lists all accessible structure members.
- **SEE ALSO** gives sources for further information.

Every driver **MUST** include `<sys/ddi.h>` and `<sys/sunddi.h>`, in that order, and last.

The following table summarizes the STREAMS structures described in this section.

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
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_IA	Solaris IA DDI
ddi_dma_req	Solaris DDI
ddi_dmae_req	Solaris IA 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
scsi_arq_status	Solaris DDI
scsi_device	Solaris DDI
scsi_extended_sense	Solaris DDI
scsi_hba_tran	Solaris DDI
scsi_inquiry	Solaris DDI

Structure	Type
<code>scsi_pkt</code>	Solaris DDI
<code>scsi_status</code>	Solaris DDI
<code>uio</code>	DDI/DKI

NOTES Do not declare arrays of structures as the size of the structures may 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.

Data Structures for Drivers

aio_req(9S)

NAME	aio_req – asynchronous I/O request structure
SYNOPSIS	<pre>#include <sys/uio.h> #include <sys/aio_req.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)
DESCRIPTION	An aio_req structure describes an asynchronous I/O request.
STRUCTURE MEMBERS	<pre>struct uio*aio_uio; /* uio structure describing the I/O request */</pre> 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	<pre>#include <sys/ddi.h> #include <sys/sunddi.h></pre>
INTERFACE LEVEL DESCRIPTION	<p>Architecture independent level 1 (DDI/DKI).</p> <p>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.</p> <p>It is important to note that a buffer header may 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.</p> <p>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.</p> <p>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.</p>
STRUCTURE MEMBERS	<pre>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 device */ size_t b_resid; /* # of bytes not transferred */ size_t b_bufsize; /* size of allocated 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 */</pre> <p>The members of the buffer header available to test or set by a driver are as follows:</p> <p>B_BUSY indicates the buffer is in use. The driver may not change this flag unless it allocated the buffer with getrbuf(9F) and no I/O operation is in progress.</p> <p>B_DONE indicates the data transfer has completed. This flag is read-only.</p> <p>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.</p>

buf(9S)

<code>B_PAGEIO</code>	indicates the buffer is being used in a paged I/O request. See the description of the <code>b_un.b_addr</code> field for more information. This flag is read-only.
<code>B_PHYS</code>	indicates the buffer header is being used for physical (direct) I/O to a user data area. See the description of the <code>b_un.b_addr</code> field for more information. This flag is read-only.
<code>B_READ</code>	indicates data is to be read from the peripheral device into main memory.
<code>B_WRITE</code>	indicates the data is to be transferred from main memory to the peripheral device. <code>B_WRITE</code> is a pseudo flag and cannot be directly tested; it is only detected as the NOT form of <code>B_READ</code> .

`b_flags` stores the buffer status and tells 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.

Valid flags are as follows:

`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, may 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 may 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 may 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` may 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.

cb_ops(9S)

NAME	cb_ops – character/block entry points structure
SYNOPSIS	<pre>#include <sys/conf.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	<p>cb_ops contains all entry points for drivers that support both character and block entry points. All leaf device drivers supporting direct user process access to a device should declare a cb_ops structure.</p> <p>All drivers which safely allow multiple threads of execution in the driver at the same time must set the D_MP flag in the cb_flag field.</p> <p>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.</p> <p>mt-streams(9F) describes other flags that may be set in the cb_flag field.</p> <p>cb_rev is the cb_ops structure revision number. This field must be set to CB_REV.</p> <p>Non-STREAMS drivers should set cb_str to NULL.</p> <p>The following DDI/DKI or DKI-only or DDI-only functions are provided in the character/block driver operations structure.</p>

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
c	XXdevmap	DDI(Sun)
c	XXmmap	DKI
c	XXsegmap	DKI
c	XXchpoll	DDI/DKI

STRUCTURE MEMBERS

block/char	Function	Description
c	XXprop_op	DDI(Sun)
c	XXaread	DDI(Sun)
c	XXawrite	DDI(Sun)
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;intcb_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

copyreq(9S)

NAME	copyreq – STREAMS data structure for the M_COPYIN and the M_COPYOUT message types
SYNOPSIS	<pre>#include <sys/stream.h></pre>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	The data structure for the M_COPYIN and the M_COPYOUT message types.
STRUCTURE MEMBERS	<pre>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; /* see below */ 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 */ /* cq_flag values */ #define STRCANON 0x01 /* b_cont data block contains */ /* canonical format specifier */ #define RECOPY 0x02 /* perform I_STR copyin again, */ /* this time using canonical */ /* format specifier */</pre>
SEE ALSO	<i>STREAMS Programming Guide</i>

NAME	copyresp – STREAMS data structure for the M_IOCTLDATA 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_IOCTLDATA message type.
STRUCTURE MEMBERS	<pre> 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 */ mbk_t *cp_private; /* private state information */ caddr_t cp_rval; /* status of request: 0 -> success; /*non-zero -> failure */ </pre>
SEE ALSO	<i>STREAMS Programming Guide</i>

datab(9S)

NAME	datab – STREAMS message data structure
SYNOPSIS	<pre>#include <sys/stream.h></pre>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	<p>The <code>datab</code> 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 <code>msgb</code> (message block) structure includes a field that points to a <code>datab</code> structure.</p> <p>A data block can have more than one message block pointing to it at one time, so the <code>db_ref</code> member keeps track of a data block's references, preventing it from being deallocated until all message blocks are finished with it.</p>
STRUCTURE MEMBERS	<pre>unsigned char *db_base; /* first byte of buffer */ unsigned char *db_lim; /* last byte (+1) of buffer */ dbref_t db_ref; /* # of message pointers to this data */ unsigned char db_type; /* message type */</pre> <p>A <code>datab</code> structure is defined as type <code>dbl_k_t</code>.</p>
SEE ALSO	<p><code>free_rtn(9S)</code>, <code>msgb(9S)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

NAME	ddi_device_acc_attr – data access attributes structure												
SYNOPSIS	<pre>#include <sys/ddi.h> #include <sys/sunddi.h></pre>												
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	<pre>ushort_t devacc_attr_version; uchar_t devacc_attr_endian_flags; uchar_t devacc_attr_dataorder;</pre> <p>The devacc_attr_version member identifies the version number of this structure. The current version number is DDI_DEVICE_ATTR_V0.</p> <p>The devacc_attr_endian_flags member describes the endian characteristics of the device. Specify one of the following values.</p> <table border="0"> <tr> <td>DDI_NEVERSWAP_ACC</td> <td>Ddata access with no byte swapping.</td> </tr> <tr> <td>DDI_STRUCTURE_BE_ACC</td> <td>Structural data access in big endian format.</td> </tr> <tr> <td>DDI_STRUCTURE_LE_ACC</td> <td>Structural data access in little endian format.</td> </tr> </table> <p>DDI_STRUCTURE_BE_ACC and DDI_STRUCTURE_LE_ACC describes the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their buses, there are examples of devices with I/O an processor that has opposite endian characteristics of the buses. When DDI_STRUCTURE_BE_ACC or DDI_STRUCTURE_LE_ACC is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.</p> <p>When DDI_NEVERSWAP_ACC is specified, byte swapping will not be invoked in the data access functions.</p> <p>The devacc_attr_dataorder member describes order in which the CPU will reference data. Specify one of the following values.</p> <table border="0"> <tr> <td>DDI_STRICTORDER_ACC</td> <td>The data references must be issued by a CPU in program order. Strict ordering is the default behavior.</td> </tr> <tr> <td>DDI_UNORDERED_OK_ACC</td> <td>The CPU may re-order the data references. This includes all kinds of re-ordering. For example, . a load followed by a store may be replaced by a store followed by a load.</td> </tr> <tr> <td>DDI_MERGING_OK_ACC</td> <td>The CPU may merge individual stores to consecutive locations. For example, the CPU</td> </tr> </table>	DDI_NEVERSWAP_ACC	Ddata 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_STRICTORDER_ACC	The data references must be issued by a CPU in program order. Strict ordering is the default behavior.	DDI_UNORDERED_OK_ACC	The CPU may re-order the data references. This includes all kinds of re-ordering. For example, . a load followed by a store may be replaced by a store followed by a load.	DDI_MERGING_OK_ACC	The CPU may merge individual stores to consecutive locations. For example, the CPU
DDI_NEVERSWAP_ACC	Ddata 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_STRICTORDER_ACC	The data references must be issued by a CPU in program order. Strict ordering is the default behavior.												
DDI_UNORDERED_OK_ACC	The CPU may re-order the data references. This includes all kinds of re-ordering. For example, . a load followed by a store may be replaced by a store followed by a load.												
DDI_MERGING_OK_ACC	The CPU may merge individual stores to consecutive locations. For example, the CPU												

ddi_device_acc_attr(9S)

may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. DDI_MERGING_OK_ACC also implies re-ordering.

DDI_LOADCACHING_OK_ACC

The CPU may 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 re-ordering.

DDI_STORECACHING_OK_ACC

The CPU may 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 re-ordering.

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.

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;

/*
```

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

```

    * 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   */
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. We assume that several apertures are grouped under one single "reg" entry. For example, the sample device has four different apertures each 32K 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 128K with each 32K 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);

```

ddi_device_acc_attr(9S)

EXAMPLE 2 Accessing a Device with Different Apertures (Continued)

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.

```
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);
```

SEE ALSO ddi_getw(9F), ddi_putw(9F), ddi_regs_map_setup(9F)

Writing Device Drivers

NAME	ddi_dma_attr – DMA attributes structure
SYNOPSIS	<pre>#include <sys/ddidmareq.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	A <code>ddi_dma_attr_t</code> structure describes device and DMA engine specific attributes necessary to allocate DMA resources for a device. The driver may have to extend the attributes with bus specific information depending on the bus to which the device is connected.
STRUCTURE MEMBERS	<pre>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 */</pre> <p><code>dma_attr_version</code> stores the version number of this DMA attribute structure. It should be set to <code>DMA_ATTR_V0</code>.</p> <p>The <code>dma_attr_addr_lo</code> and <code>dma_attr_addr_hi</code> fields specify the address range the device's DMA engine can access. The <code>dma_attr_addr_lo</code> field describes the inclusive lower 64-bit boundary. The <code>dma_attr_addr_hi</code> describes the inclusive upper 64-bit boundary. The system will ensure that allocated DMA resources are within the range specified. See <code>ddi_dma_cookie(9S)</code>.</p> <p>The <code>dma_attr_count_max</code> describes an inclusive upper bound for the device's DMA counter register. For example, <code>0xFFFFFFFF</code> 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.</p> <p>The <code>dma_attr_align</code> specifies alignment requirements for allocated DMA resources. This field can be used to force more restrictive alignment than imposed by <code>dma_attr_burstsizes</code> or <code>dma_attr_minxfer</code>, such as alignment at a page boundary. Most drivers will set this to 1 indicating byte alignment.</p> <p>Note that <code>dma_attr_align</code> only specifies alignment requirements for allocated DMA resources. The buffer passed to <code>ddi_dma_addr_bind_handle(9F)</code> or <code>ddi_dma_buf_bind_handle(9F)</code> must have an equally restrictive alignment (see <code>ddi_dma_mem_alloc(9F)</code>).</p>

ddi_dma_attr(9S)

The `dma_attr_burstsizes` field describes the possible burst sizes the device's DMA engine can accept. The format of the data sizes is binary encoded in terms of powers of two. When DMA resources are allocated, the system may 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 may 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 will allocate DMA resources for the device such 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 64K boundary. DMA resource allocation functions may 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 device's DMA scatter/gather list. Possible values are as follows:

- < 0 Device DMA engine is not constrained by the size – for example, DMA chaining.
- = 0 Reserved.
- = 1 Device DMA engine does not support scatter/gather such as third party DMA.
- > 1 Device DMA engine uses scatter/gather. `dma_attr_sgllen` 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, a single segment's size will be a multiple of the device granularity. Or if `dma_attr_sgllen` is larger than 1 within a window, the sum of the sizes for a subgroup of segments will be a multiple of the device granularity.

Note that 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:

`DDI_DMA_FORCE_PHYSICAL`

Some platforms, such as SPARC systems, support what is called DVMA (Direct Virtual Memory Access). On these platforms the device is provided with a virtual

ddi_dma_attr(9S)

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 support in addition 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)` will be `DDI_DMA_BADATTR`. In this case, the driver has to clear `DDI_DMA_FORCE_PHYSICAL` and retry the operation.

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

The corresponding `ddi_dma_attr_t` structure would be 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)0xffffffff, /* 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 */
};
```

SEE ALSO `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_dma_cookie(9S)`

Writing Device Drivers

ddi_dma_cookie(9S)

NAME	ddi_dma_cookie – DMA address cookie
SYNOPSIS	#include <sys/sunddi.h>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	The <code>ddi_dma_cookie_t</code> structure contains DMA address information required to program a DMA engine. It is filled in by a call to <code>ddi_dma_getwin(9F)</code> , <code>ddi_dma_addr_bind_handle(9F)</code> , or <code>ddi_dma_buf_bind_handle(9F)</code> to get device specific DMA transfer information for a DMA request or a DMA window.
STRUCTURE MEMBERS	<pre>uint64_t dmac_laddress; /* 64 bit address */ uint32_t dmac_address; /* 32 bit address */ size_t dmac_size; /* transfer size */ uint_t dmac_type; /* bus specific type bits */</pre> <p><code>dmac_laddress</code> 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. <code>dmac_address</code> specifies a 32-bit I/O address. It should be used for devices which 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 <code>ddi_dma_attr(9S)</code> structure.</p> <p><code>dmac_size</code> describes the length of the transfer in bytes.</p> <p><code>dmac_type</code> contains bus specific type bits, if appropriate. For example, a device on a VME bus will have VME address modifier bits placed here.</p>
SEE ALSO	<code>pci(4)</code> , <code>sbus(4)</code> , <code>sysbus(4)</code> , <code>ddi_dma_addr_bind_handle(9F)</code> , <code>ddi_dma_buf_bind_handle(9F)</code> , <code>ddi_dma_getwin(9F)</code> , <code>ddi_dma_nextcookie(9F)</code> , <code>ddi_dma_attr(9S)</code>

Writing Device Drivers

NAME	ddi_dmae_req – DMA engine request structure
SYNOPSIS	<pre>#include <sys/dma_engine.h></pre>
INTERFACE LEVEL	Solaris IA DDI specific (Solaris IA DDI).
DESCRIPTION	A <code>ddi_dmae_req</code> structure is used by a device driver to describe the parameters for a DMA channel. This structure contains all the information necessary to set up the channel, except for the DMA memory address and transfer count. The defaults as specified below support most standard devices. Other modes may be desirable for some devices, or to increase performance. The DMA engine request structure is passed to <code>ddi_dmae_prog(9F)</code> .
STRUCTURE MEMBERS	<p>The <code>ddi_dmae_req</code> 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.</p> <pre>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 */</pre> <p><code>der_command</code> specifies what DMA operation is to be performed. The value <code>DMAE_CMD_WRITE</code> signifies that data is to be transferred from memory to the I/O device. The value <code>DMAE_CMD_READ</code> signifies that data is to be transferred from the I/O device to memory. This field must be set by the driver before calling <code>ddi_dmae_prog()</code>.</p> <p><code>der_bufprocess</code> On some bus types, a driver may set <code>der_bufprocess</code> to the value <code>DMAE_BUF_CHAIN</code> to specify that multiple DMA cookies will be given to the DMA engine for a single I/O transfer, thus effecting a scatter/gather operation. In this mode of operation, the driver calls <code>ddi_dmae_prog()</code> to give the DMA engine the DMA engine request structure and a pointer to the first cookie. The <code>proc</code> structure member must be set to the address of a driver <code>nextcookie</code> routine that takes one argument, specified by the <code>procparms</code> structure member, and returns a pointer to a structure of type <code>ddi_dma_cookie_t</code> 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 <code>proc</code> structure member to obtain the next cookie from the driver. The driver's <code>nextcookie</code> 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 <code>(*proc)()</code> must return the <code>NULL</code> pointer.</p> <p>A driver may only specify the <code>DMAE_BUF_CHAIN</code> flag if the particular bus architecture supports the use of multiple DMA cookies in a single I/O transfer. A bus DMA engine may support this feature either with a fixed-length scatter/gather list, or via an interrupt chaining feature such as the one implemented in the EISA architecture. A driver must ascertain whether its parent bus nexus supports this</p>

ddi_dmae_req(9S)

feature by examining the scatter/gather list size returned in the `dlim_sgllen` member of the DMA limit structure (see `ddi_dma_lim_IA(9S)`) returned by the driver's call to `ddi_dmae_getlim()`. If the size of the scatter/gather list is 1, then no chaining is available, the driver must not specify the `DMAE_BUF_CHAIN` flag in the `ddi_dmae_req` structure it passes to `ddi_dmae_prog()`, and the driver need not provide a `nextcookie` routine.

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

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

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

As described above, a driver making use of DMA chaining must prefetch DMA cookies before calling `ddi_dmae_prog()`. There are two reasons why the driver must do this. First, the driver must have some way to know the total I/O count with which to program the I/O device. This I/O count must match the total size of all the DMA segments that will be chained together into one DMA operation. Depending on the size of the scatter/gather list and the memory position and alignment of the DMA object, all or just part of the current DMA window may be able to participate in a single I/O operation. The driver must compute the I/O count by adding up the sizes of the prefetched DMA cookies. The number of cookies whose sizes are to be summed is the lesser of (a) the size of the scatter/gather list, or (b) the number of segments remaining in the window. Second, on some bus architectures, the driver's `nextcookie` routine may be called

from a high-level interrupt routine. If the cookies were not prefetched, the `nextcookie` routine would have to call `ddi_dma_nextseg()` and `ddi_dma_segtocookie()` from a high-level interrupt routine, which is not recommended.

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

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

`der_path`

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

`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_SINGL` in the `der_trans` structure member. On EISA buses, these other timing modes are available:

<code>DMAE_CYCLES_2</code>	specifies type "A" timing;
<code>DMAE_CYCLES_3</code>	specifies type "B" timing;
<code>DMAE_CYCLES_4</code>	specifies "Burst" timing.

`der_trans`

specifies the bus transfer mode that the DMA engine should expect from the device. The default value of zero (`DMAE_TRANS_SINGL`) specifies that the device will perform one transfer for each bus arbitration cycle. Devices that use ISA compatible timing (specified by a value of zero, which is the default, in the `der_cycles` structure member) should use the `DMAE_TRANS_SINGL` mode. On EISA buses, a `der_trans` value of `DMAE_TRANS_BLOCK` specifies that the device will perform a block of transfers for each arbitration cycle. A value of `DMAE_TRANS_DMND` specifies that the device will perform the Demand Transfer Mode protocol.

ddi_dmae_req(9S)

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	IA

SEE ALSO `eisa(4)`, `isa(4)`, `attributes(5)`, `ddi_dma_segtocookie(9F)`, `ddi_dmae(9F)`, `ddi_dma_lim_IA(9S)`, `ddi_dma_req(9S)`

NAME	ddi_dma_lim_sparc, ddi_dma_lim – SPARC DMA limits structure
SYNOPSIS	<pre>#include <sys/ddidmareq.h></pre>
INTERFACE LEVEL	Solaris SPARC DDI specific (Solaris SPARC DDI).
DESCRIPTION	A <code>ddi_dma_lim</code> 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	<pre>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_cntr_max; /* inclusive upper bound of dma engine's */ /* address limit */ uint_t dlim_burstsizes; /* binary encoded dma burst sizes */ uint_t dlim_minxfer; /* minimum effective dma transfer size */ uint_t dlim_dmaspeed; /* average dma data rate (kb/s) */</pre> <p>The <code>dlim_addr_lo</code> and <code>dlim_addr_hi</code> fields specify the address range the device's DMA engine can access. The <code>dlim_addr_lo</code> field describes the lower 32 bit boundary of the device's DMA engine, the <code>dlim_addr_hi</code> describes the inclusive upper 32 bit boundary. The system will allocate DMA resources in a way that the address for programming the device's DMA engine (see <code>ddi_dma_cookie(9S)</code> or <code>ddi_dma_htoc(9F)</code>) will be within this range. For example, if your device can access the whole 32 bit address range, you may use <code>[0,0xFFFFFFFF]</code>. If your device has just a 16 bit address register but will access the top of the 32 bit address range, then <code>[0xFFFF0000,0xFFFFFFFF]</code> would be the right limit.</p> <p>The <code>dlim_cntr_max</code> 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 simply a latch rather than a full register. For example, the upper 8 bits of a 32 bit address register may be a latch. This splits the address register into a portion which acts as a true address register (24 bits) for a 16 megabyte segment and a latch (8 bits) to hold a segment number. To describe these limits, you would specify <code>0xFFFF</code> in the <code>dlim_cntr_max</code> structure.</p> <p>The <code>dlim_burstsizes</code> 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 would be <code>0x17</code>. 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 <code>burstsizes</code> value may be modified. Prior to enabling DMA for the specific device, the driver that owns the DMA engine should check (using <code>ddi_dma_burstsizes(9F)</code>) what the allowed burst sizes have become and program the DMA engine appropriately.</p>

ddi_dma_lim_sparc(9S)

The `dlim_minxfer` field describes the minimum effective DMAtransfer size (in units of bytes). It must be a power of two. This value specifies the minimum effective granularity of the DMAengine. It is distinct from `dlim_burstsizes` in that it describes the minimum amount of access a DMAtransfer will effect.

`dlim_burstsizes` describes in what electrical fashion the DMAengine might perform its accesses, while `dlim_minxfer` describes the minimum amount of memory that can be touched by the DMAtransfer. As a resource request is handled by the system, the `dlim_minxfer` value may be modified contingent upon the presence (and use) of I/O caches and DMA write buffers in between the DMAengine and the object that DMAis 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 DMAengine (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 may be left as zero. Its intended use is to provide some hints about how much DMAresources this device may need.

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_IA(9S)`, `ddi_dma_req(9S)`

NAME	ddi_dma_lim_x86 – IA DMA limits structure
SYNOPSIS	<pre>#include <sys/ddidmareq.h></pre>
INTERFACE LEVEL	Solaris IA DDI specific (Solaris IA DDI)
DESCRIPTION	<p>A <code>ddi_dma_lim</code> 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 will be broken up, if necessary, into multiple sub-requests, each of which conforms to the limitations expressed in the <code>ddi_dma_lim</code> structure.</p> <p>This structure should be filled in by calling the routine <code>ddi_dmae_getlim(9F)</code> which 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 may <i>further restrict</i> some of the values in the structure members. A driver should take care to not <i>relax</i> any restrictions imposed by <code>ddi_dmae_getlim()</code>.</p>
STRUCTURE MEMBERS	<pre>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 */</pre> <p>The <code>dlim_addr_lo</code> and <code>dlim_addr_hi</code> fields specify the address range the device's DMA engine can access. The <code>dlim_addr_lo</code> field describes the lower 32 bit boundary of the device's DMA engine; <code>dlim_addr_hi</code> describes the inclusive upper 32 bit boundary. The system will allocate DMA resources in a way that the address for programming the device's DMA engine (see <code>ddi_dma_cookie(9S)</code> or <code>ddi_dma_segtocookie(9F)</code>) will be within this range. For example, if your device can access the whole 32 bit address range, you may use <code>[0, 0xFFFFFFFF]</code>.</p> <p>The <code>dlim_minxfer</code> 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 describes the minimum amount of memory that can be touched by the DMA transfer. As a resource request is handled by the system, the <code>dlim_minxfer</code> value may 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 retrieved using <code>ddi_dma_devalign(9F)</code>.</p> <p>The <code>dlim_version</code> field specifies the version number of this structure. This field should be set to <code>DMALIM_VER0</code>.</p>

ddi_dma_lim_x86(9S)

The `dlim_adreg_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 simply a latch rather than a full register. For example, the upper 16 bits of a 32 bit address register may be a latch. This splits the address register into a portion which 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 would 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. It is used as a bit mask, so 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 will be 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 will not be smaller than the minimum transfer value, but may be less than the granularity; however the total transfer length of the scatter/gather list will be a multiple of the granularity value.

The `dlim_sgllen` field specifies the maximum number of entries in the scatter/gather list. It 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, this field should be set 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, this field should be set 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).
DESCRIPTION	A <code>ddi_dma_req</code> structure describes a request for DMA resources. A driver may use it to describe forms of and ways to allocate DMA resources for a DMA request.
STRUCTURE MEMBERS	<pre> ddi_dma_lim_t *dmar_limits; /* Caller's dma engine's */ /* constraints */ uint_t dmar_flags; /* Contains information for */ /* mapping routines */ int (*dmar_fp)(caddr_t); /* Callback function */ caddr_t dmar_arg; /* Callback function's argument */ ddi_dma_obj_t dmar_object; /* Description of the object */ /* to be mapped */ </pre> <p>For the definition of the DMA limits structure, which <code>dmar_limits</code> points to, see <code>ddi_dma_lim_sparc(9S)</code> or <code>ddi_dma_lim_IA(9S)</code>.</p> <p>Valid values for <code>dmar_flags</code> are:</p> <pre> DDI_DMA_WRITE /* Direction memory --> IO */ DDI_DMA_READ /* Direction IO --> memory */ DDI_DMA_RDWR /* Both read and write */ DDI_DMA_REDZONE /* Establish an MMU redzone at end of mapping */ DDI_DMA_PARTIAL /* Partial mapping is allowed */ DDI_DMA_CONSISTENT /* Byte consistent access wanted */ DDI_DMA_SBUS_64BIT /* Use 64 bit capability on SBus */ </pre> <p><code>DDI_DMA_WRITE</code>, <code>DDI_DMA_READ</code> and <code>DDI_DMA_RDWR</code> describe the intended direction of the DMA transfer. Some implementations may explicitly disallow <code>DDI_DMA_RDWR</code>.</p> <p><code>DDI_DMA_REDZONE</code> asks the system to establish a protected <i>red zone</i> after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support it.</p> <p><code>DDI_DMA_PARTIAL</code> tells the system that the caller can accept a partial mapping. That is, if the size of the object exceeds the resources available, only allocate a portion of the object and return status indicating so. At a later point, the caller can use <code>ddi_dma_curwin(9F)</code> and <code>ddi_dma_movwin(9F)</code> to change the valid portion of the object that has resources allocated.</p> <p><code>DDI_DMA_CONSISTENT</code> gives a hint to the system that the object should be mapped for <i>byte consistent</i> access. Normal data transfers usually use a <i>streaming</i> mode of operation. They start at a specific point, transfer a fairly large amount of data sequentially, and then stop usually on a aligned boundary. Control mode data transfers for memory resident device control blocks (for example ethernet message descriptors) do not access memory in such a sequential fashion. Instead, they tend to modify a few words or bytes, move around and maybe modify a few more. There are many machine implementations that make this difficult to control in a generic and</p>

ddi_dma_req(9S)

seamless fashion. Therefore, explicit synchronization steps using `ddi_dma_sync(9F)` or `ddi_dma_free(9F)` are required in order to make the view of a memory object shared between a CPU and a DMA device consistent. However, proper use of the `DDI_DMA_CONSISTENT` flag gives a hint to the system so that it will attempt to pick resources such that these synchronization steps are as efficient as possible.

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

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

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

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

dmар_object Structure

The `dmар_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 are allocated for DMA.

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

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

```
struct v_address virt_obj;
```

It is specified as:

```
struct v_address {
    caddr_t    v_addr;    /* base virtual address */
    struct as  *v_as;     /* pointer to address space */
    void      *v_priv;   /* priv data for shadow I/O */
};
```

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

ddi_idevice_cookie(9S)

NAME	ddi_idevice_cookie – device interrupt cookie
SYNOPSIS	<pre>#include <sys/ddi.h> #include <sys/sunddi.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	The <code>ddi_idevice_cookie_t</code> structure contains interrupt priority and interrupt vector information for a device. This structure is useful for devices having programmable bus-interrupt levels. <code>ddi_add_intr(9F)</code> assigns values to the <code>ddi_idevice_cookie_t</code> structure members.
STRUCTURE MEMBERS	<pre>u_short idev_vector; /* interrupt vector */ ushort_t idev_priority; /* interrupt priority */</pre> <p>The <code>idev_vector</code> field contains the interrupt vector number for vectored bus architectures such as VMEbus. The <code>idev_priority</code> field contains the bus interrupt priority level.</p>
SEE ALSO	<code>ddi_add_intr(9F)</code> <i>Writing Device Drivers</i>

NAME	ddi_mapdev_ctl – device mapping-control structure
SYNOPSIS	<pre>#include <sys/conf.h> #include <sys/devops.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	<p>Future releases of Solaris will provide this structure for binary and source compatibility. However, for increased functionality, use <code>devmap_callback_ctl(9S)</code> instead. See <code>devmap_callback_ctl(9S)</code> for details.</p> <p>A <code>ddi_mapdev_ctl</code> structure describes a set of routines that allow a device driver to manage events on mappings of the device created by <code>ddi_mapdev(9F)</code>.</p> <p>See <code>mapdev_access(9E)</code>, <code>mapdev_dup(9E)</code> and <code>mapdev_free(9E)</code> for more details on these entry points.</p>
STRUCTURE MEMBERS	<pre>int mapdev_rev; int (*mapdev_access)(ddi_mapdev_handle_t handle, void *devprivate, off_t offset); void (*mapdev_free)(ddi_mapdev_handle_t handle, void *devprivate); int (*mapdev_dup)(ddi_mapdev_handle_t handle, void *devprivate, ddi_mapdev_handle_t new_handle, void **new_devprivate);</pre> <p>A device driver should allocate the device mapping control structure and initialize the following fields:</p> <p><code>mapdev_rev</code> Must be set to <code>MAPDEV_REV</code>.</p> <p><code>mapdev_access</code> Must be set to the address of the <code>mapdev_access(9E)</code> entry point.</p> <p><code>mapdev_free</code> Must be set to the address of the <code>mapdev_free(9E)</code> entry point.</p> <p><code>mapdev_dup</code> Must be set to the address of the <code>mapdev_dup(9E)</code> entry point.</p>
SEE ALSO	<p><code>exit(2)</code>, <code>fork(2)</code>, <code>mmap(2)</code>, <code>munmap(2)</code>, <code>mapdev_access(9E)</code>, <code>mapdev_dup(9E)</code>, <code>mapdev_free(9E)</code>, <code>segmap(9E)</code>, <code>ddi_mapdev(9F)</code>, <code>ddi_mapdev_intercept(9F)</code>, <code>ddi_mapdev_nointercept(9F)</code></p>

Writing Device Drivers

devmap_callback_ctl(9S)

NAME	devmap_callback_ctl – device mapping-control structure										
SYNOPSIS	#include <sys/ddidevmap.h>										
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).										
DESCRIPTION	<p>A <code>devmap_callback_ctl</code> 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 <code>devmap_setup(9F)</code> or <code>ddi_devmap_segmap(9F)</code>.</p> <p>Device drivers pass the initialized <code>devmap_callback_ctl</code> structure to either <code>devmap_devmem_setup(9F)</code> or <code>devmap_umem_setup(9F)</code> in the <code>devmap(9E)</code> entry point during the mapping setup. The system will make a private copy of the structure for later use. Device drivers may specify different <code>devmap_callback_ctl</code> for different mappings.</p> <p>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:</p> <table><tr><td><code>devmap_rev</code></td><td>Version number. Set this to <code>DEVMAP_OPS_REV</code></td></tr><tr><td><code>devmap_map</code></td><td>Set to the address of the <code>devmap_map(9E)</code> entry point or to <code>NULL</code> if the driver does not support this callback. If set, the system will call the <code>devmap_map(9E)</code> entry point during the <code>mmap(2)</code> 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.</td></tr><tr><td><code>devmap_access</code></td><td>Set to the address of the <code>devmap_access(9E)</code> entry point or to <code>NULL</code> if the driver does not support this callback. If set, the system will call the driver's <code>devmap_access(9E)</code> entry point during memory access. The system expects <code>devmap_access(9E)</code> to call either <code>devmap_do_ctxmgt(9F)</code> or <code>devmap_default_access(9F)</code> to load the memory address translations before it returns to the system.</td></tr><tr><td><code>devmap_dup</code></td><td>Set to the address of the <code>devmap_dup(9E)</code> entry point or to <code>NULL</code> if the driver does not support this call. If set, the system will call the <code>devmap_dup(9E)</code> entry point during the <code>fork(2)</code> system call.</td></tr><tr><td><code>devmap_unmap</code></td><td>Set to the address of the <code>devmap_unmap(9E)</code> entry point or to <code>NULL</code> if the driver does not support this call. If set, the system will call the <code>devmap_unmap(9E)</code> entry point during the <code>munmap(2)</code> or <code>exit(2)</code> system calls.</td></tr></table>	<code>devmap_rev</code>	Version number. Set this to <code>DEVMAP_OPS_REV</code>	<code>devmap_map</code>	Set to the address of the <code>devmap_map(9E)</code> entry point or to <code>NULL</code> if the driver does not support this callback. If set, the system will call the <code>devmap_map(9E)</code> entry point during the <code>mmap(2)</code> 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.	<code>devmap_access</code>	Set to the address of the <code>devmap_access(9E)</code> entry point or to <code>NULL</code> if the driver does not support this callback. If set, the system will call the driver's <code>devmap_access(9E)</code> entry point during memory access. The system expects <code>devmap_access(9E)</code> to call either <code>devmap_do_ctxmgt(9F)</code> or <code>devmap_default_access(9F)</code> to load the memory address translations before it returns to the system.	<code>devmap_dup</code>	Set to the address of the <code>devmap_dup(9E)</code> entry point or to <code>NULL</code> if the driver does not support this call. If set, the system will call the <code>devmap_dup(9E)</code> entry point during the <code>fork(2)</code> system call.	<code>devmap_unmap</code>	Set to the address of the <code>devmap_unmap(9E)</code> entry point or to <code>NULL</code> if the driver does not support this call. If set, the system will call the <code>devmap_unmap(9E)</code> entry point during the <code>munmap(2)</code> or <code>exit(2)</code> system calls.
<code>devmap_rev</code>	Version number. Set this to <code>DEVMAP_OPS_REV</code>										
<code>devmap_map</code>	Set to the address of the <code>devmap_map(9E)</code> entry point or to <code>NULL</code> if the driver does not support this callback. If set, the system will call the <code>devmap_map(9E)</code> entry point during the <code>mmap(2)</code> 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.										
<code>devmap_access</code>	Set to the address of the <code>devmap_access(9E)</code> entry point or to <code>NULL</code> if the driver does not support this callback. If set, the system will call the driver's <code>devmap_access(9E)</code> entry point during memory access. The system expects <code>devmap_access(9E)</code> to call either <code>devmap_do_ctxmgt(9F)</code> or <code>devmap_default_access(9F)</code> to load the memory address translations before it returns to the system.										
<code>devmap_dup</code>	Set to the address of the <code>devmap_dup(9E)</code> entry point or to <code>NULL</code> if the driver does not support this call. If set, the system will call the <code>devmap_dup(9E)</code> entry point during the <code>fork(2)</code> system call.										
<code>devmap_unmap</code>	Set to the address of the <code>devmap_unmap(9E)</code> entry point or to <code>NULL</code> if the driver does not support this call. If set, the system will call the <code>devmap_unmap(9E)</code> entry point during the <code>munmap(2)</code> or <code>exit(2)</code> system calls.										
STRUCTURE MEMBERS	<pre>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,</pre>										

devmap_callback_ctl(9S)

```
        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

dev_ops(9S)

NAME	dev_ops – device operations structure																						
SYNOPSIS	<pre>#include <sys/conf.h> #include <sys/devops.h></pre>																						
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).																						
DESCRIPTION	<p>dev_ops contains driver common fields and pointers to the bus_ops and cb_ops(9S).</p> <p>Following are the device functions provided in the device operations structure. All fields must be set at compile time.</p> <table border="0"> <tr> <td style="padding-right: 20px;">devo_rev</td> <td>Driver build version. Set this to DEVO_REV.</td> </tr> <tr> <td>devo_refcnt</td> <td>Driver reference count. Set this to 0.</td> </tr> <tr> <td>devo_getinfo</td> <td>Get device driver information (see getinfo(9E)).</td> </tr> <tr> <td>devo_identify</td> <td>Determine if a driver is associated with a device. See identify(9E).</td> </tr> <tr> <td>devo_probe</td> <td>Probe device. See probe(9E).</td> </tr> <tr> <td>devo_attach</td> <td>Attach driver to dev_info. See attach(9E).</td> </tr> <tr> <td>devo_detach</td> <td>Detach/prepare driver to unload. See detach(9E).</td> </tr> <tr> <td>devo_reset</td> <td>Reset device. Not supported in this release.) Set this to nodev.</td> </tr> <tr> <td>devo_cb_ops</td> <td>Pointer to cb_ops(9S) structure for leaf drivers.</td> </tr> <tr> <td>devo_bus_ops</td> <td>Pointer to bus operations structure for nexus drivers. Set this to NULL if this is for a leaf driver.</td> </tr> <tr> <td>devo_power</td> <td>Power a device attached to be system. See power(9E).</td> </tr> </table>	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	Determine if a driver is associated with a device. See identify(9E).	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 be system. See power(9E).
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	Determine if a driver is associated with a device. See identify(9E).																						
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 be system. See power(9E).																						
STRUCTURE MEMBERS	<pre>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);</pre>																						
SEE ALSO	attach(9E), detach(9E), getinfo(9E), identify(9E), probe(9E), power(9E), nodev(9F)																						
	<i>Writing Device Drivers</i>																						

NAME	fmodsw – STREAMS module declaration structure
SYNOPSIS	<pre>#include <sys/stream.h> #include <sys/conf.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)
DESCRIPTION	<p>The fmodsw structure contains information for STREAMS modules. All STREAMS modules must define a fmodsw structure.</p> <p>f_name must match mi_idname in the module_info structure. See module_info(9S).</p> <p>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.</p>
STRUCTURE MEMBERS	<pre>char f_name[FMNAMESZ + 1]; /* module name */ struct streamtab *f_str; /* streams information */ int f_flag; /* flags */</pre>
SEE ALSO	<p>mt-streams(9F), modlstrmod(9S), module_info(9S)</p> <p><i>STREAMS Programming Guide</i></p>

free_rtn(9S)

NAME	free_rtn – structure that specifies a driver’s message freeing routine
SYNOPSIS	<pre>#include <sys/stream.h></pre>
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	<pre>void (*free_func)() /* user’s freeing routine */ char *free_arg /* arguments to free_func() */</pre> <p>The free_rtn structure is defined as type frtn_t.</p>
SEE ALSO	esballoc(9F), freeb(9F), datab(9S) <i>STREAMS Programming Guide</i>

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*

iovec(9S)

NAME	iovec – data storage structure for I/O using uio
SYNOPSIS	<pre>#include <sys/uio.h></pre>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	An <code>iovec</code> structure describes a data storage area for transfer in a <code>uio(9S)</code> structure. Conceptually, it may be thought of as a base address and length specification.
STRUCTURE MEMBERS	<pre>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 */</pre>
SEE ALSO	<code>uio(9S)</code> <i>Writing Device Drivers</i>

NAME	kstat – kernel statistics structure												
SYNOPSIS	<pre>#include <sys/types.h> #include <sys/kstat.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>												
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)												
DESCRIPTION	<p>Each kernel statistic (<i>kstat</i>) exported by device drivers consists of a header section and a data section. The <i>kstat</i> structure is the header portion of the statistic.</p> <p>A driver receives a pointer to a <i>kstat</i> structure from a successful call to <i>kstat_create(9F)</i>. Drivers should never allocate a <i>kstat</i> structure in any other manner.</p> <p>After allocation, the driver should perform any further initialization needed before calling <i>kstat_install(9F)</i> to actually export the <i>kstat</i>.</p>												
STRUCTURE MEMBERS	<pre>void *ks_data; /* kstat type-specific data */ ulong_t ks_ndata; /* # of type-specific 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 this kstat's data */</pre> <p>The members of the <i>kstat</i> structure available to examine or set by a driver are as follows:</p> <table border="0"> <tr> <td style="vertical-align: top;"><i>ks_data</i></td> <td>Points to the data portion of the <i>kstat</i>. Either allocated by <i>kstat_create(9F)</i> for the drivers use, or by the driver if it is using virtual kstats.</td> </tr> <tr> <td style="vertical-align: top;"><i>ks_ndata</i></td> <td>The number of data records in this <i>kstat</i>. Set by the <i>ks_update(9E)</i> routine.</td> </tr> <tr> <td style="vertical-align: top;"><i>ks_data_size</i></td> <td>The amount of data pointed to by <i>ks_data</i>. Set by the <i>ks_update(9E)</i> routine.</td> </tr> <tr> <td style="vertical-align: top;"><i>ks_update</i></td> <td>Is a pointer to a routine which dynamically updates <i>kstat</i>. This is useful for drivers where the underlying device keeps cheap hardware stats, but extraction is expensive. Instead of constantly keeping the <i>kstat</i> data section up to date, the driver can supply a <i>ks_update(9E)</i> function which updates the <i>kstat</i> data section on demand. To take advantage of this feature, set the <i>ks_update</i> field before calling <i>kstat_install(9F)</i>.</td> </tr> <tr> <td style="vertical-align: top;"><i>ks_private</i></td> <td>Is a private field for the driver's use. Often used in <i>ks_update(9E)</i>.</td> </tr> <tr> <td style="vertical-align: top;"><i>ks_lock</i></td> <td>Is a pointer to a mutex that protects this <i>kstat</i>. <i>kstat</i> data sections are optionally protected by the per-<i>kstat</i> <i>ks_lock</i>. If <i>ks_lock</i> is non-NULL, <i>kstat</i> clients (such as <i>/dev/kstat</i>) will</td> </tr> </table>	<i>ks_data</i>	Points to the data portion of the <i>kstat</i> . Either allocated by <i>kstat_create(9F)</i> for the drivers use, or by the driver if it is using virtual kstats.	<i>ks_ndata</i>	The number of data records in this <i>kstat</i> . Set by the <i>ks_update(9E)</i> routine.	<i>ks_data_size</i>	The amount of data pointed to by <i>ks_data</i> . Set by the <i>ks_update(9E)</i> routine.	<i>ks_update</i>	Is a pointer to a routine which dynamically updates <i>kstat</i> . This is useful for drivers where the underlying device keeps cheap hardware stats, but extraction is expensive. Instead of constantly keeping the <i>kstat</i> data section up to date, the driver can supply a <i>ks_update(9E)</i> function which updates the <i>kstat</i> data section on demand. To take advantage of this feature, set the <i>ks_update</i> field before calling <i>kstat_install(9F)</i> .	<i>ks_private</i>	Is a private field for the driver's use. Often used in <i>ks_update(9E)</i> .	<i>ks_lock</i>	Is a pointer to a mutex that protects this <i>kstat</i> . <i>kstat</i> data sections are optionally protected by the per- <i>kstat</i> <i>ks_lock</i> . If <i>ks_lock</i> is non-NULL, <i>kstat</i> clients (such as <i>/dev/kstat</i>) will
<i>ks_data</i>	Points to the data portion of the <i>kstat</i> . Either allocated by <i>kstat_create(9F)</i> for the drivers use, or by the driver if it is using virtual kstats.												
<i>ks_ndata</i>	The number of data records in this <i>kstat</i> . Set by the <i>ks_update(9E)</i> routine.												
<i>ks_data_size</i>	The amount of data pointed to by <i>ks_data</i> . Set by the <i>ks_update(9E)</i> routine.												
<i>ks_update</i>	Is a pointer to a routine which dynamically updates <i>kstat</i> . This is useful for drivers where the underlying device keeps cheap hardware stats, but extraction is expensive. Instead of constantly keeping the <i>kstat</i> data section up to date, the driver can supply a <i>ks_update(9E)</i> function which updates the <i>kstat</i> data section on demand. To take advantage of this feature, set the <i>ks_update</i> field before calling <i>kstat_install(9F)</i> .												
<i>ks_private</i>	Is a private field for the driver's use. Often used in <i>ks_update(9E)</i> .												
<i>ks_lock</i>	Is a pointer to a mutex that protects this <i>kstat</i> . <i>kstat</i> data sections are optionally protected by the per- <i>kstat</i> <i>ks_lock</i> . If <i>ks_lock</i> is non-NULL, <i>kstat</i> clients (such as <i>/dev/kstat</i>) will												

kstat(9S)

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, so 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*)`; it is declared as `(void*)` in the `kstat` header so that users don't 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	<pre>#include <sys/types.h> #include <sys/kstat.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>										
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)										
DESCRIPTION	<p>Interrupt statistics are kept in the <code>kstat_intr</code> structure. When <code>kstat_create(9F)</code> creates an interrupt <code>kstat</code>, the <code>ks_data</code> field is a pointer to one of these structures. The macro <code>KSTAT_INTR_PTR()</code> is provided to retrieve this field. It looks like this:</p> <pre>#define KSTAT_INTR_PTR(kptr) ((kstat_intr_t *) (kptr)->ks_data)</pre> <p>An interrupt is a hard interrupt (sourced from the hardware device itself), a soft interrupt (induced by the system via 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).</p> <p>Drivers generally only report claimed hard interrupts and soft interrupts from their handlers, but measurement of the spurious class of interrupts is useful for autovectoring devices in order to pinpoint any interrupt latency problems in a particular system configuration.</p> <p>Devices that have more than one interrupt of the same type should use multiple structures.</p>										
STRUCTURE MEMBERS	<pre>ulong_t intrs[KSTAT_NUM_INTRS]; /* interrupt counters */</pre> <p>The only member exposed to drivers is the <code>intrs</code> 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:</p> <table border="0"> <tr> <td><code>KSTAT_INTR_HARD</code></td> <td>Hard interrupt.</td> </tr> <tr> <td><code>KSTAT_INTR_SOFT</code></td> <td>Soft interrupt.</td> </tr> <tr> <td><code>KSTAT_INTR_WATCHDOG</code></td> <td>Watchdog interrupt.</td> </tr> <tr> <td><code>KSTAT_INTR_SPURIOUS</code></td> <td>Spurious interrupt.</td> </tr> <tr> <td><code>KSTAT_INTR_MULTSVC</code></td> <td>Multiple service interrupt.</td> </tr> </table>	<code>KSTAT_INTR_HARD</code>	Hard interrupt.	<code>KSTAT_INTR_SOFT</code>	Soft interrupt.	<code>KSTAT_INTR_WATCHDOG</code>	Watchdog interrupt.	<code>KSTAT_INTR_SPURIOUS</code>	Spurious interrupt.	<code>KSTAT_INTR_MULTSVC</code>	Multiple service interrupt.
<code>KSTAT_INTR_HARD</code>	Hard interrupt.										
<code>KSTAT_INTR_SOFT</code>	Soft interrupt.										
<code>KSTAT_INTR_WATCHDOG</code>	Watchdog interrupt.										
<code>KSTAT_INTR_SPURIOUS</code>	Spurious interrupt.										
<code>KSTAT_INTR_MULTSVC</code>	Multiple service interrupt.										
SEE ALSO	<p><code>kstat(9S)</code></p> <p><i>Writing Device Drivers</i></p>										

kstat_io(9S)

NAME	kstat_io – structure for I/O kstats
SYNOPSIS	<pre>#include <sys/types.h> #include <sys/kstat.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)
DESCRIPTION	<p>I/O kstat statistics are kept in a <code>kstat_io</code> structure. When <code>kstat_create(9F)</code> creates an I/O kstat, the <code>ks_data</code> field is a pointer to one of these structures. The macro <code>KSTAT_IO_PTR()</code> is provided to retrieve this field. It looks like this:</p> <pre>#define KSTAT_IO_PTR(kptr) ((kstat_io_t *) (kptr)->ks_data)</pre>
STRUCTURE MEMBERS	<pre>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 */</pre> <p>The <code>nread</code> field should be updated by the driver with the number of bytes successfully read upon completion.</p> <p>The <code>nwritten</code> field should be updated by the driver with the number of bytes successfully written upon completion.</p> <p>The <code>reads</code> field should be updated by the driver after each successful read operation.</p> <p>The <code>writes</code> field should be updated by the driver after each successful write operation</p> <p>Other I/O statistics are updated through the use of the <code>kstat_queue(9F)</code> functions.</p>
SEE ALSO	<p><code>kstat_create(9F)</code>, <code>kstat_named_init(9F)</code>, <code>kstat_queue(9F)</code>, <code>kstat_runq_back_to_waitq(9F)</code>, <code>kstat_runq_enter(9F)</code>, <code>kstat_runq_exit(9F)</code>, <code>kstat_waitq_enter(9F)</code>, <code>kstat_waitq_exit(9F)</code>, <code>kstat_waitq_to_runq(9F)</code></p> <p><i>Writing Device Drivers</i></p>

NAME	kstat_named – structure for named kstats
SYNOPSIS	<pre>#include <sys/types.h> #include <sys/kstat.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>
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. They are returned as an array pointed to by the ks_data field.
STRUCTURE MEMBERS	<pre>union { char c[16]; long l; ulong_t ul; longlong_t ll; u_longlong_t ull; } value; /* value of counter */</pre> <p>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().</p>
SEE ALSO	kstat_create(9F), kstat_named_init(9F) <i>Writing Device Drivers</i>

linkblk(9S)

NAME	linkblk – STREAMS data structure sent to multiplexor drivers to indicate a link
SYNOPSIS	<pre>#include <sys/stream.h></pre>
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	<pre>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. */</pre>
SEE ALSO	ioctl(2), streamio(7I) <i>STREAMS Programming Guide</i>

NAME	modldrv – linkage structure for loadable drivers
SYNOPSIS	<code>#include <sys/modctl.h></code>
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	<pre> struct mod_ops *drv_modops; char *drv_link_info; struct dev_ops *drv_dev_ops; </pre> <p>drv_modops Must always be initialized to the address of mod_driverops. This identifies the module as a loadable driver.</p> <p>drv_linkinfo Can be any string up to MODMAXNAMELEN, and is used to describe the module. This is usually the name of the driver, but can contain other information such as a version number.</p> <p>drv_dev_ops Pointer to the driver's dev_ops(9S) structure.</p>
SEE ALSO	add_drv(1M), dev_ops(9S), modlinkage(9S) <i>Writing Device Drivers</i>

modlinkage(9S)

NAME	modlinkage – module linkage structure
SYNOPSIS	<pre>#include <sys/modctl.h></pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)
DESCRIPTION	The modlinkage structure is provided by the module writer to the routines which install, remove, and retrieve information from a module. See <code>_init(9E)</code> , <code>_fini(9E)</code> and <code>_info(9E)</code> .
STRUCTURE MEMBERS	<pre>int ml_rev void *ml_linkage[4];</pre> <p><code>ml_rev</code> Is the revision of the loadable modules system. This must have the value <code>MODREV_1</code>.</p> <p><code>ml_linkage</code> Is a null-terminated array of pointers to linkage structures. For driver modules there is only one linkage structure.</p>
SEE ALSO	<code>add_drv(1M)</code> , <code>_fini(9E)</code> , <code>_info(9E)</code> , <code>_init(9E)</code> , <code>modldrv(9S)</code> , <code>modlstrmod(9S)</code> <i>Writing Device Drivers</i>

NAME	modlstrmod – linkage structure for loadable STREAMS modules
SYNOPSIS	<code>#include <sys/modctl.h></code>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)
DESCRIPTION	The <code>modlstrmod</code> structure is used by STREAMS modules to export module specific information to the kernel.
STRUCTURE MEMBERS	<pre> struct mod_ops *strmod_modops; char *strmod_linkinfo; struct fmodsw *strmod_fmodsw; </pre> <p><code>strmod_modops</code> Must always be initialized to the address of <code>mod_strmodops</code>. This identifies the module as a loadable STREAMS module.</p> <p><code>strmod_linkinfo</code> Can be any string up to <code>MODMAXNAMELEN</code>, and is used to describe the module. This is usually the name of the module, but can contain other information (such as a version number).</p> <p><code>strmod_fmodsw</code> 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.</p>
SEE ALSO	<p><code>modload(1M)</code></p> <p><i>Writing Device Drivers</i></p>

module_info(9S)

NAME	module_info – STREAMS driver identification and limit value structure
SYNOPSIS	#include <sys/stream.h>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	<p>When a module or driver is declared, several identification and limit values can be set. These values are stored in the <code>module_info</code> structure.</p> <p>The <code>module_info</code> structure is intended to be read-only. However, the flow control limits (<code>mi_hiwat</code> and <code>mi_lowat</code>) and the packet size limits (<code>mi_minpsz</code> and <code>mi_maxpsz</code>) are copied to the <code>QUEUE</code> structure, where they may be modified.</p>
STRUCTURE MEMBERS	<pre>ushort_t mi_idnum; /* module ID number */ char *mi_idname; /* module name */ ssize_t mi_minpsz; /* maximum packet size */ size_t mi_hiwat; /* high water mark */ size_t mi_lowat; /* low water mark */</pre> <p>The constant <code>FMNAMESZ</code>, limiting the length of a module's name, is set to eight in this release.</p>
SEE ALSO	<p>queue(9S)</p> <p><i>STREAMS Programming Guide</i></p>

NAME	msgb – STREAMS message block structure
SYNOPSIS	<code>#include <sys/stream.h></code>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	<p>A STREAMS message is made up of one or more message blocks, referenced by a pointer to a msgb structure. The <code>b_next</code> and <code>b_prev</code> pointers are used to link messages together on a QUEUE. The <code>b_cont</code> pointer links message blocks together when a message is composed of more than one block.</p> <p>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.</p>
STRUCTURE MEMBERS	<pre> 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 */ </pre> <p>Valid flags are as follows:</p> <p>MSGMARK Last byte of message is marked.</p> <p>MSGDELIM Message is delimited.</p> <p>The msgb structure is defined as type <code>mblk_t</code>.</p>
SEE ALSO	<p>datab(9S)</p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>

qband(9S)

NAME	qband – STREAMS queue flow control information structure
SYNOPSIS	#include <sys/stream.h>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	<p>The qband structure contains flow control information for each priority band in a queue.</p> <p>The qband structure is defined as type qband_t.</p>
STRUCTURE MEMBERS	<pre>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 */</pre> <p>Valid flags are as follows:</p> <p>QB_FULL Band is considered full.</p> <p>QB_WANTW Someone wants to write to band.</p>
SEE ALSO	strqget(9F), strqset(9F), msgb(9S), queue(9S)
	<i>STREAMS Programming Guide</i>
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 may 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	<code>#include <sys/stream.h></code>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	The <code>qinit</code> structure contains pointers to processing procedures for a <code>QUEUE</code> . The <code>streamtab</code> structure for the module or driver contains pointers to one <code>queue(9S)</code> structure for both upstream and downstream processing.
STRUCTURE MEMBERS	<pre> 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 */ </pre>
SEE ALSO	<p><code>queue(9S)</code>, <code>streamtab(9S)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>
NOTES	This release includes no support for module statistics.

queclass(9S)

NAME	queclass – a STREAMS macro that returns the queue message class definitions for a given message block
SYNOPSIS	<pre>#include <sys/stream.h> queclass(mblk_t *bp);</pre>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	<p>queclass returns the queue message class definition for a given data block pointed to by the message block <i>bp</i> passed in.</p> <p>The message may either be QNORM, a normal priority message, or QPCTL, a high priority message.</p>
SEE ALSO	<i>STREAMS Programming Guide</i>

NAME	queue – STREAMS queue structure														
SYNOPSIS	<code>#include <sys/stream.h></code>														
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).														
DESCRIPTION	<p>A STREAMS driver or module consists of two queue structures, one for upstream processing (read) and one for downstream processing (write). This structure is the major building block of a stream. It contains pointers to the processing procedures, pointers to the next and previous queues in the stream, flow control parameters, and a pointer defining the position of its messages on the STREAMS scheduler list.</p> <p>The queue structure is defined as type <code>queue_t</code>.</p>														
STRUCTURE MEMBERS	<pre> struct qinit*q_qinfo; /* module or driver entry points */ struct msgb*q_first; /* first message in queue */ struct msgb*q_last; /* last message in queue */ struct queue*q_next; /* next queue in stream */ struct queue*q_link; /* to next queue for scheduling*/ void *q_ptr; /* pointer to private data structure */ size_t q_count; /* approximate size of message queue */ uint_t q_flag; /* status of queue */ ssize_t q_minpsz; /* smallest packet accepted by QUEUE*/ ssize_t q_maxpsz; /*largest packet accepted by QUEUE */ size_t q_hiwat; /* high water mark */ size_t q_lowat; /* low water mark */ </pre> <p>Valid flags are as follows:</p> <table border="0"> <tr> <td style="padding-right: 20px;">QENAB</td> <td>Queue is already enabled to run.</td> </tr> <tr> <td style="padding-right: 20px;">QWANTR</td> <td>Someone wants to read queue.</td> </tr> <tr> <td style="padding-right: 20px;">QWANTW</td> <td>Someone wants to write to queue.</td> </tr> <tr> <td style="padding-right: 20px;">QFULL</td> <td>Queue is considered full.</td> </tr> <tr> <td style="padding-right: 20px;">QREADR</td> <td>This is the reader (first) queue.</td> </tr> <tr> <td style="padding-right: 20px;">QUSE</td> <td>This queue in use (allocation).</td> </tr> <tr> <td style="padding-right: 20px;">QNOENB</td> <td>Do not enable queue by wasy of <code>putq()</code>.</td> </tr> </table>	QENAB	Queue is already enabled to run.	QWANTR	Someone wants to read queue.	QWANTW	Someone wants to write to queue.	QFULL	Queue is considered full.	QREADR	This is the reader (first) queue.	QUSE	This queue in use (allocation).	QNOENB	Do not enable queue by wasy of <code>putq()</code> .
QENAB	Queue is already enabled to run.														
QWANTR	Someone wants to read queue.														
QWANTW	Someone wants to write to queue.														
QFULL	Queue is considered full.														
QREADR	This is the reader (first) queue.														
QUSE	This queue in use (allocation).														
QNOENB	Do not enable queue by wasy of <code>putq()</code> .														
SEE ALSO	<p><code>strqget(9F)</code>, <code>strqset(9F)</code>, <code>module_info(9S)</code>, <code>msgb(9S)</code>, <code>qinit(9S)</code>, <code>streamtab(9S)</code></p> <p><i>Writing Device Drivers</i></p> <p><i>STREAMS Programming Guide</i></p>														

scsi_address(9S)

NAME	scsi_address – SCSI address structure
SYNOPSIS	<pre>#include <sys/scsi/scsi.h></pre>
INTERFACE LEVEL	Solaris architecture specific (Solaris DDI).
DESCRIPTION	<p>A <code>scsi_address</code> structure defines the addressing components for 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.</p> <p>The <code>pkt_address</code> member of a <code>scsi_pkt(9S)</code> is initialized by <code>scsi_init_pkt(9F)</code>.</p>
STRUCTURE MEMBERS	<pre>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 */</pre> <p><code>a_hba_tran</code> is a pointer to the controlling HBA's transport vector structure. The SCSI interface uses this field to pass any transport requests from the SCSI target device drivers to the HBA driver.</p> <p><code>a_target</code> is the target component of the SCSI address.</p> <p><code>a_lun</code> 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. The <code>makecom(9F)</code> family of functions use the <code>a_lun</code> field to set the logical unit field in the SCSI CDB, for compatibility with SCSI-1.</p>
SEE ALSO	<code>makecom(9F)</code> , <code>scsi_init_pkt(9F)</code> , <code>scsi_hba_tran(9S)</code> , <code>scsi_pkt(9S)</code> <i>Writing Device Drivers</i>

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 <code>scsi_ifsetcap(9F)</code> and the "auto-rqsense" capability, the target driver must allocate a status area in the SCSI packet structure (see <code>scsi_pkt(9S)</code>) for the auto request sense structure. In the event of a check <i>condition</i> the transport layer will automatically execute a request sense command. This 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	<pre> 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 */ </pre> <p><code>sts_status</code> is the SCSI status of the original command. If the status indicates a check <i>condition</i> then the transport layer may have performed an auto request sense command.</p> <p><code>sts_rqpkt_status</code> is the SCSI status of the request sense command.</p> <p><code>sts_rqpkt_reason</code> is the completion reason of the request sense command. If the reason is not <code>CMD_CMPLT</code>, then the request sense command did not complete normally.</p> <p><code>sts_rqpkt_resid</code> 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 <code>SENSE_LENGTH</code> bytes.</p> <p><code>sts_rqpkt_state</code> has bit positions representing the five most important status that a SCSI command can go through.</p> <p><code>sts_rqpkt_statistics</code> maintains transport-related statistics of the request sense command.</p> <p><code>sts_sensedata</code> contains the actual sense data if the request sense command completed normally.</p>
SEE ALSO	<p><code>scsi_ifgetcap(9F)</code>, <code>scsi_init_pkt(9F)</code>, <code>scsi_extended_sense(9S)</code>, <code>scsi_pkt(9S)</code></p> <p><i>Writing Device Drivers</i></p>

scsi_asc_key_strings(9S)

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 ASCQ and pointer to the related ASCII string.
STRUCTURE MEMBERS	<pre> 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</pre>
SEE ALSO	scsi_vu_errmsg(9F) <i>ANSI Small Computer System Interface-2 (SCSI-2)</i> <i>Writing Device Drivers</i>

NAME	scsi_device – SCSI device structure
SYNOPSIS	<code>#include <sys/scsi/scsi.h></code>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	<p>The <code>scsi_device</code> structure stores common information about each SCSI logical unit, including pointers to areas that contain both generic and device specific information. There is one <code>scsi_device</code> structure for each logical unit attached to the system. The host adapter driver initializes part of this structure prior to <code>probe(9E)</code> and destroys this structure after a probe failure or successful <code>detach(9E)</code>.</p>
STRUCTURE MEMBERS	<pre> struct scsi_address sd_address; /* Routing information */ dev_info_t *sd_dev; /* Cross-reference to our dev_info_t */ kmutex_t sd_mutex; /* Mutex for this device */ struct scsi_inquiry *sd_inq; /* scsi_inquiry data structure */ struct scsi_extended_sense *sd_sense; /* Optional request sense buffer ptr */ caddr_t sd_private; /* Target drivers private data */ </pre> <p><code>sd_address</code> contains the routing information that the target driver normally copies into a <code>scsi_pkt(9S)</code> structure using the collection of <code>makecom(9F)</code> functions. The SCSI library routines use this information to determine which host adapter, SCSI bus, and target/lun a command is intended for. This structure is initialized by the host adapter driver.</p> <p><code>sd_dev</code> is a pointer to the corresponding <code>dev_info</code> structure. This pointer is initialized by the host adapter driver.</p> <p><code>sd_mutex</code> 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 <code>mutex(9F)</code>.</p> <p><code>sd_inq</code> is initially <code>NULL</code> (zero). After executing <code>scsi_probe(9F)</code> this field contains the inquiry data associated with the particular device.</p> <p><code>sd_sense</code> is initially <code>NULL</code> (zero). If the target driver wants to use this field for storing <code>REQUEST SENSE</code> data, it should allocate an <code>scsi_extended_sense(9S)</code> buffer and set this field to the address of this buffer.</p> <p><code>sd_private</code> is reserved for the use of target drivers and should generally be used to point to target specific data structures.</p>
SEE ALSO	<p><code>detach(9E)</code>, <code>probe(9E)</code>, <code>makecom(9F)</code>, <code>mutex(9F)</code>, <code>scsi_probe(9F)</code>, <code>scsi_extended_sense(9S)</code>, <code>scsi_pkt(9S)</code></p> <p><i>Writing Device Drivers</i></p>

scsi_extended_sense(9S)

NAME	scsi_extended_sense – SCSI extended sense structure
SYNOPSIS	#include <sys/scsi/scsi.h>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	The <code>scsi_extended_sense</code> 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 <code>scsi_device(9S)</code> structure.
STRUCTURE MEMBERS	<pre> 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 */ </pre> <p><code>es_valid</code>, if set, indicates that the information field contains valid information.</p> <p><code>es_class</code> should be 0x7.</p> <p><code>es_code</code> is either 0x0 or 0x1.</p> <p><code>es_segnum</code> 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.</p> <p><code>es_filmk</code>, if set, indicates that the current command had read a filemark or setmark (sequential access devices only).</p> <p><code>es_eom</code>, if set, indicates that an end-of-medium condition exists (sequential access and printer devices only).</p> <p><code>es_ili</code>, if set, indicates that the requested logical block length did not match the logical block length of the data on the medium.</p> <p><code>es_key</code> indicates generic information describing an error or exception condition. The following sense keys are defined:</p> <p>KEY_NO_SENSE Indicates that there is no specific sense key information to be reported.</p>

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 may 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 a COPY, COMPARE, and COPY AND VERIFY command was aborted.

KEY_ABORTED_COMMAND

Indicates that the target aborted the command.

KEY_EQUAL

Indicates 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 may 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.

scsi_extended_sense(9S)

`es_add_len` indicates the number of additional sense bytes to follow.

`es_cmd_info` contains information that depends on the command which 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_key_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 <code>scsi_hba_tran_t</code> structure defines vectors that an HBA driver exports to SCSSA interfaces so that HBA specific functions can be executed.
STRUCTURE MEMBERS	<pre> dev_info_t *tran_hba_dip; /* HBAs dev_info pointer */ void *tran_hba_private; /* HBA softstate */ void *tran_tgt_privat /* HBA target private pointer */ struct scsi_device *tran_sd; /* scsi_device */ int (*tran_tgt_init)(); /* transport target */ int (*tran_tgt_probe)(); /* initialization */ int (*tran_tgt_free)(); /* transport target probe */ void (*tran_start)(); /* transport target free */ int (*tran_reset)(); /* transport start */ int (*tran_abort)(); /* transport reset */ int (*tran_getcap)(); /* transport abort */ int (*tran_setcap)(); /* capability retrieval */ struct scsi_pkt (*tran_init_pkt)(); /* capability establishment */ void (*tran_destroy_pkt)(); /* packet and dma allocation */ void (*tran_dmafree)(); /* packet and dma */ void (*tran_sync_pkt)(); /* deallocation */ void (*tran_reset_notify)(); /* dma deallocation */ int (*tran_bus_reset)(); /* sync DMA */ int (*tran_quiesce)(); /* bus reset notification */ int (*tran_unquiesce)(); /* reset bus only */ </pre> <p><code>tran_hba_dip</code> <code>dev_info</code> pointer to the HBA supplying the <code>scsi_hba_tran</code> structure.</p> <p><code>tran_hba_private</code> Private pointer which the HBA driver can use to refer to the device's soft state structure.</p> <p><code>tran_tgt_private</code> Private pointer which the HBA can use to refer to per-target specific data. This field may only be used when the <code>SCSI_HBA_TRAN_CLONE</code> flag is specified in <code>scsi_hba_attach(9F)</code>. In this case, the HBA driver must initialize this field in its <code>tran_tgt_init(9E)</code> entry point.</p> <p><code>tran_sd</code> Pointer to <code>scsi_device(9S)</code> structure if cloning; otherwise NULL.</p> <p><code>tran_tgt_init</code> Is the function entry allowing per-target HBA initialization, if necessary.</p> <p><code>tran_tgt_probe</code> Is the function entry allowing per-target <code>scsi_probe(9F)</code> customization, if necessary.</p> <p><code>tran_tgt_free</code> Is the function entry allowing per-target HBA deallocation, if necessary.</p>

scsi_hba_tran(9S)

<code>tran_start</code>	Is the function entry that starts a SCSI command execution on the HBA hardware.
<code>tran_reset</code>	Is the function entry that resets a SCSI bus or target device.
<code>tran_abort</code>	Is the function entry that aborts one SCSI command, or all pending SCSI commands.
<code>tran_getcap</code>	Is the function entry that retrieves a SCSI capability.
<code>tran_setcap</code>	Is the function entry that sets a SCSI capability.
<code>tran_init_pkt</code>	Is the function entry that allocates a <code>scsi_pkt</code> structure.
<code>tran_destroy_pkt</code>	Is the function entry that frees a <code>scsi_pkt</code> structure allocated by <code>tran_init_pkt</code> .
<code>tran_dmafree</code>	is the function entry that frees DMA resources which were previously allocated by <code>tran_init_pkt</code> .
<code>tran_sync_pkt</code>	Synchronize data in <i>pkt</i> after a data transfer has been completed.
<code>tran_reset_notify</code>	Is the function entry allowing a target to register a bus reset notification request with the HBA driver.
<code>tran_bus_reset</code>	Is the function entry that resets the SCSI bus without resetting targets.
<code>tran_quiesce</code>	Is the function entry that waits for all outstanding commands to complete and blocks (or queues) any I/O requests issued.
<code>tran_unquiesce</code>	Is the function entry that allows I/O activities to resume on the SCSI bus.

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 <code>scsi_inquiry</code> 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 <code>scsi_device(9S)</code> structure and typically filled in by <code>scsi_probe(9F)</code> .																						
STRUCTURE MEMBERS	<pre> uchar_t inq_dtype; /* peripheral qualifier, device type */ uchar_t inq_rmb :1; /* removable media */ uchar_t inq_qual :7; /* device type qualifier */ uchar_t inq_iso :2; /* ISO version */ uchar_t inq_ecma :3; /* ANSI version */ uchar_t inq_aenc :1; /* async event notification cap. */ uchar_t inq_trmiop :1; /* supports TERMINATE I/O PROC msg */ uchar_t inq_rdf :4; /* response data format */ uchar_t inq_len; /* additional length */ uchar_t inq_reladdr :1; /* supports relative addressing */ uchar_t inq_wbus32 :1; /* supports 32 bit wide data xfers */ uchar_t inq_wbus16 :1; /* supports 16 bit wide data xfers */ uchar_t inq_sync :1; /* 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 */ </pre> <p><code>inq_dtype</code> 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:</p> <table border="0"> <tr> <td><code>DTYPE_ARRAY_CTRL</code></td> <td>Array controller device (for example, RAID).</td> </tr> <tr> <td><code>DTYPE_DIRECT</code></td> <td>Direct-access device (for example, magnetic disk).</td> </tr> <tr> <td><code>DTYPE_ESI</code></td> <td>Enclosure services device.</td> </tr> <tr> <td><code>DTYPE_SEQUENTIAL</code></td> <td>Sequential-access device (for example, magnetic tape).</td> </tr> <tr> <td><code>DTYPE_PRINTER</code></td> <td>Printer device.</td> </tr> <tr> <td><code>DTYPE_PROCESSOR</code></td> <td>Processor device.</td> </tr> <tr> <td><code>DTYPE_WORM</code></td> <td>Write-once device (for example, some optical disks).</td> </tr> <tr> <td><code>DTYPE_RODIRECT</code></td> <td>CD-ROM device.</td> </tr> <tr> <td><code>DTYPE_SCANNER</code></td> <td>Scanner device.</td> </tr> <tr> <td><code>DTYPE_OPTICAL</code></td> <td>Optical memory device (for example, some optical disks).</td> </tr> <tr> <td><code>DTYPE_CHANGER</code></td> <td>Medium Changer device (for example, jukeboxes).</td> </tr> </table>	<code>DTYPE_ARRAY_CTRL</code>	Array controller device (for example, RAID).	<code>DTYPE_DIRECT</code>	Direct-access device (for example, magnetic disk).	<code>DTYPE_ESI</code>	Enclosure services device.	<code>DTYPE_SEQUENTIAL</code>	Sequential-access device (for example, magnetic tape).	<code>DTYPE_PRINTER</code>	Printer device.	<code>DTYPE_PROCESSOR</code>	Processor device.	<code>DTYPE_WORM</code>	Write-once device (for example, some optical disks).	<code>DTYPE_RODIRECT</code>	CD-ROM device.	<code>DTYPE_SCANNER</code>	Scanner device.	<code>DTYPE_OPTICAL</code>	Optical memory device (for example, some optical disks).	<code>DTYPE_CHANGER</code>	Medium Changer device (for example, jukeboxes).
<code>DTYPE_ARRAY_CTRL</code>	Array controller device (for example, RAID).																						
<code>DTYPE_DIRECT</code>	Direct-access device (for example, magnetic disk).																						
<code>DTYPE_ESI</code>	Enclosure services device.																						
<code>DTYPE_SEQUENTIAL</code>	Sequential-access device (for example, magnetic tape).																						
<code>DTYPE_PRINTER</code>	Printer device.																						
<code>DTYPE_PROCESSOR</code>	Processor device.																						
<code>DTYPE_WORM</code>	Write-once device (for example, some optical disks).																						
<code>DTYPE_RODIRECT</code>	CD-ROM device.																						
<code>DTYPE_SCANNER</code>	Scanner device.																						
<code>DTYPE_OPTICAL</code>	Optical memory device (for example, some optical disks).																						
<code>DTYPE_CHANGER</code>	Medium Changer device (for example, jukeboxes).																						

scsi_inquiry(9S)

DTYPE_COMM	Communications 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 shall also use 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.

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 PROCESS message.

inq_rdf, if reset, indicates the INQUIRY data format is as specified in SCSI-1.

`inq_inq_len` is the additional length field which specifies the length in bytes of the parameters.

`inq_reladdr`, if set, indicates that the device supports the relative addressing mode of this logical unit.

`inq_wbus32`, if set, indicates that the device supports 32-bit wide data transfers.

`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 queuing.

`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.

SEE ALSO `scsi_probe(9F)`, `scsi_device(9S)`

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

Writing Device Drivers

scsi_pkt(9S)

NAME	scsi_pkt – SCSI packet structure
SYNOPSIS	#include <sys/scsi/scsi.h>
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI).
DESCRIPTION	A <code>scsi_pkt</code> structure defines the packet which is allocated by <code>scsi_init_pkt(9F)</code> . The target driver fills in some information, and passes it to <code>scsi_transport(9F)</code> for execution on the target. The HBA fills in some 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	<pre> 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 */ </pre> <p><code>pkt_ha_private</code> An opaque pointer which the Host Bus Adapter uses to reference a private data structure used to transfer <code>scsi_pkt</code> requests.</p> <p><code>pkt_address</code> Initialized by <code>scsi_init_pkt(9F)</code> and serves to record the intended route and recipient of a request.</p> <p><code>pkt_private</code> Reserved for the use of the target driver and is not changed by the HBA driver.</p> <p><code>pkt_comp</code> 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</p>

	will call the function pointed to by this field and pass a pointer to the packet as argument. The callback routine itself is called from interrupt context and must not sleep nor call any function which may sleep.
<code>pkt_flags</code>	Provides additional information about how the target driver wants the command to be executed. See <code>pkt_flag</code> Definitions.
<code>pkt_time</code>	Will be set by the target driver to represent the maximum length of time in seconds that this command is allowed take to complete. <code>pkt_time</code> may be 0 if no timeout is required.
<code>pkt_scbp</code>	Points to either a struct <code>scsi_status(9S)</code> or, if <code>auto-rqsense</code> is enabled, and <code>pkt_state</code> includes <code>STATE_ARQ_DONE</code> , a struct <code>scsi_arq_status</code> . If <code>scsi_status</code> is returned, the SCSI status byte resulting from the requested command is available; if <code>scsi_arq_status(9S)</code> is returned, the sense information is also available.
<code>pkt_cdbp</code>	Points to a kernel addressable buffer whose length was specified by a call to the proper resource allocation routine, <code>scsi_init_pkt(9F)</code> .
<code>pkt_resid</code>	Contains a residual count, either the number of data bytes that have not been transferred (<code>scsi_transport(9F)</code>) or the number of data bytes for which DMA resources could not be allocated <code>scsi_init_pkt(9F)</code> . In the latter case, partial DMA resources may only be allocated if <code>scsi_init_pkt(9F)</code> is called with the <code>PKT_DMA_PARTIAL</code> flag.
<code>pkt_state</code>	Has bit positions representing the six most important states that a SCSI command can go through (see <code>pkt_state</code> Definitions).
<code>pkt_statistics</code>	Maintains some transport-related statistics. (see <code>pkt_statistics</code> Definitions).
<code>pkt_reason</code>	Contains a completion code that indicates why the <code>pkt_comp</code> function was called.

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

**pkt_flags
Definitions:**

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

scsi_pkt(9S)

FLAG_NOINTR	Run command with no command completion callback; command is complete upon return from <code>scsi_transport(9F)</code> .
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	This command is a request sense command.
FLAG_HEAD	This command should be put at the head of the queue.

**pkt_reason
Definitions:**

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

CMD_CMPLT	No transport errors—normal completion.
CMD_INCOMPLETE	Transport stopped with abnormal state.
CMD_DMA_DERR	DMA direction 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.
pkt_state Definitions:	The definitions that are appropriate for the structure member <code>pkt_state</code> 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 cmd.
pkt_statistics Definitions:	The definitions that are appropriate for the structure member <code>pkt_statistics</code> 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	<code>tran_init_pkt(9E)</code> , <code>scsi_arq_status(9S)</code> , <code>scsi_init_pkt(9F)</code> , <code>scsi_transport(9F)</code> , <code>scsi_status(9S)</code> <i>Writing Device Drivers</i>	

scsi_status(9S)

NAME	scsi_status – SCSI status structure												
SYNOPSIS	#include <sys/scsi/scsi.h>												
INTERFACE LEVEL	Solaris DDI specific (Solaris DDI)												
DESCRIPTION	The SCSI-2 standard defines a status byte which is normally sent by the target to the initiator during the status phase at the completion of each command.												
STRUCTURE MEMBERS	<pre>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 */ uchar sts_chk :1; /* check condition */</pre> <p><code>sts_chk</code> indicates that a contingent allegiance condition has occurred.</p> <p><code>sts_cm</code> is returned whenever the requested operation is satisfied</p> <p><code>sts_busy</code> 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 at a later time.</p> <p><code>sts_is</code> 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 <code>scsi_ifsetcap(9F)</code>). If <code>sts_is</code> and <code>sts_busy</code> are both set, then a reservation conflict has occurred.</p> <p><code>sts_scsi2</code> is the SCSI-2 modifier bit. If <code>sts_scsi2</code> and <code>sts_chk</code> are both set, this indicates a command terminated status. If <code>sts_scsi2</code> and <code>sts_busy</code> are both set, this indicates that the command queue in the target is full.</p> <p>For accessing the status as a byte, the following values are appropriate:</p> <table><tr><td><code>STATUS_GOOD</code></td><td>This status indicates that the target has successfully completed the command.</td></tr><tr><td><code>STATUS_CHECK</code></td><td>This status indicates that a contingent allegiance condition has occurred.</td></tr><tr><td><code>STATUS_MET</code></td><td>This status is returned when the requested operations are satisfied.</td></tr><tr><td><code>STATUS_BUSY</code></td><td>This status indicates that the target is busy.</td></tr><tr><td><code>STATUS_INTERMEDIATE</code></td><td>This status is returned for every successfully completed command in a series of linked commands.</td></tr><tr><td><code>STATUS_SCSI2</code></td><td>This is the SCSI-2 modifier bit.</td></tr></table>	<code>STATUS_GOOD</code>	This status indicates that the target has successfully completed the command.	<code>STATUS_CHECK</code>	This status indicates that a contingent allegiance condition has occurred.	<code>STATUS_MET</code>	This status is returned when the requested operations are satisfied.	<code>STATUS_BUSY</code>	This status indicates that the target is busy.	<code>STATUS_INTERMEDIATE</code>	This status is returned for every successfully completed command in a series of linked commands.	<code>STATUS_SCSI2</code>	This is the SCSI-2 modifier bit.
<code>STATUS_GOOD</code>	This status indicates that the target has successfully completed the command.												
<code>STATUS_CHECK</code>	This status indicates that a contingent allegiance condition has occurred.												
<code>STATUS_MET</code>	This status is returned when the requested operations are satisfied.												
<code>STATUS_BUSY</code>	This status indicates that the target is busy.												
<code>STATUS_INTERMEDIATE</code>	This status is returned for every successfully completed command in a series of linked commands.												
<code>STATUS_SCSI2</code>	This is the SCSI-2 modifier bit.												

scsi_status(9S)

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

streamtab(9S)

NAME	streamtab – STREAMS entity declaration structure
SYNOPSIS	<pre>#include <sys/stream.h></pre>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	<p>Each STREAMS driver or module must have a <code>streamtab</code> structure.</p> <p><code>streamtab</code> is made up of <code>qinit</code> structures for both the read and write queue portions of each module or driver. Multiplexing drivers require both upper and lower <code>qinit</code> structures. The <code>qinit</code> structure contains the entry points through which the module or driver routines are called.</p> <p>Normally, the read <code>QUEUE</code> contains the <code>open</code> and <code>close</code> routines. Both the read and write queue can contain <code>put</code> and service procedures.</p>
STRUCTURE MEMBERS	<pre>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*/</pre>
SEE ALSO	<p><code>qinit(9S)</code></p> <p><i>STREAMS Programming Guide</i></p>

NAME	stroptions – options structure for M_SETOPTS message
SYNOPSIS	<pre>#include <sys/stream.h> #include <sys/stropts.h> #include <sys/ddi.h> #include <sys/sunddi.h></pre>
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	<pre>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 */</pre> <p>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</p> <pre>SO_READOPT Set read option. SO_WROFF Set write offset. SO_MINPSZ Set min packet size SO_MAXPSZ Set max 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_NDELOFF Old TTY semantics for NDELAY reads/writes. SO_NDELOFFSTREAMS Semantics for NDELAY reads/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 stream. SO_BAND Water marks affect band. SO_ERROPT Set error option.</pre>

stroptions(9S)

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

<code>RNORM</code>	Read msg norm.
<code>RMSGD</code>	Read msg discard.
<code>RMSGN</code>	Read msg 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:

<code>RERRNORM</code>	Persistent read errors; default.
<code>RERRNONPERSIST</code>	Non-persistent read errors.

OR'ed with either none or one of:

<code>WERRNORM</code>	Persistent write errors; default.
<code>WERRNONPERSIST</code>	Non-persistent write errors.

SEE ALSO `read(2)`, `streamio(7I)`

STREAMS Programming Guide

NAME	tuple – Card Information Structure (CIS) access structure
SYNOPSIS	<code>#include <sys/pccard.h></code>
INTERFACE LEVEL	Solaris DDI Specific (Solaris DDI)
DESCRIPTION	<p>The <code>tuple_t</code> 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.</p> <p>The CIS information allows PC Cards to be self-identifying, meaning that the CIS provides information to the system so that it can identify the proper PC Card driver for the PC Card, and configuration information so that the driver can allocate appropriate resources to configure the PC Card for proper operation in the system.</p> <p>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.</p> <p>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.</p> <p>To examine a PC Card's CIS, first a PC Card driver must locate the desired tuple by calling <code>csx_GetFirstTuple(9F)</code>. Once the first tuple is located, subsequent tuples may be located by calling <code>csx_GetNextTuple(9F)</code>. See <code>csx_GetFirstTuple(9F)</code>. 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.</p> <p>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 <code>csx_GetTupleData(9F)</code>.</p> <p>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.</p>
STRUCTURE MEMBERS	<p>The structure members of <code>tuple_t</code> are:</p> <pre> 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]; </pre>

tuple(9S)

```
/* 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 via the <code>csx_ParseTuple(9F)</code> function if set.
DesiredTuple	This field is the requested tuple type code to be returned, when calling <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> . <code>RETURN_FIRST_TUPLE</code> is used to return the first tuple regardless of tuple type, if it exists. <code>RETURN_NEXT_TUPLE</code> 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 <code>csx_GetTupleData(9F)</code> .
TupleDataMax	This field is the size of the tuple data buffer that Card Services uses to return raw tuple data from <code>csx_GetTupleData(9F)</code> . 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 <code>csx_GetTupleData(9F)</code> .
TupleData	This field is an array of bytes containing the raw tuple data body contents returned by <code>csx_GetTupleData(9F)</code> .
TupleCode	This field is the tuple type code and is returned by <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> when a tuple matching the <code>DesiredTuple</code> field is returned.
TupleLink	This field is the tuple link, the offset to the next tuple, and is returned by <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> when a tuple matching the <code>DesiredTuple</code> field is returned.

tuple(9S)

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

uio(9S)

NAME	uio – scatter/gather I/O request structure
SYNOPSIS	#include <sys/uio.h>
INTERFACE LEVEL	Architecture independent level 1 (DDI/DKI).
DESCRIPTION	<p>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/written.</p> <p>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.</p> <p>uio structures allocated by the driver should be initialized to zero before use, by bzero(9F), kmem_zalloc(9F), or an equivalent.</p>
STRUCTURE MEMBERS	<pre> iovec_t *uio_iov; /* pointer to the start of the iovec */ /* list for the uio structure */ int uio_iovcnt; /* the number of iovecs in the list */ off_t uio_offset; /* 32-bit offset into file where data is */ /* transferred from or to. See NOTES. */ offset_t uio_loffset; /* 64-bit offset into file where data is */ /* transferred from or to. See NOTES. */ uio_seg_t uio_segflg; /* identifies the type of I/O transfer: */ /* UIO_SYSSPACE: kernel <-> kernel */ /* UIO_USERSPACE: kernel <-> user */ short uio_fmode; /* file mode flags (not driver settable) */ daddr_t uio_limit; /* 32-bit ulimit for file (maximum block */ /* offset). not driver settable. See NOTES. */ diskaddr_t uio_llimit; /* 64-bit ulimit for file (maximum block */ /* offset). not driver settable. See NOTES. */ int uio_resid; /* residual count */ </pre> <p>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.</p>
SEE ALSO	<p>aread(9E), awrite(9E), read(9E), write(9E), bzero(9F), kmem_zalloc(9F), uiomove(9F), cb_ops(9S), iovec(9S)</p> <p><i>Writing Device Drivers</i></p>
NOTES	<p>Only one of 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.</p> <p>Only one of 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.</p>

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.

uio(9S)

Index

A

aio_req — asynchronous I/O request structure, 16
asynchronous I/O request structure — aio_req, 16

B

buf — block I/O data transfer structure, 17

C

Card Information Structure (CIS) access structure — tuple, 89
character/block entry points structure for drivers, — cb_ops, 20
copyreq — STREAMS data structure for the M_COPYIN and the M_COPYOUT message types, 22
copyresp — STREAMS data structure for the M_IOCTLDATA message type, 23

D

data access attributes structure — ddi_device_acc_attr, 25
ddi_device_acc_attr — data access attributes structure, 25

DDI device mapping

ddi_mapdev_ctl — device mapping-control structure, 45
devmap_callback_ctl — device mapping-control structure, 46

DDI direct memory access

DMA limits structure — ddi_dma_lim, 37, 39
DMA cookie structure — ddi_dma_cookie, 32
DMA Request structure — ddi_dma_req, 41
ddi_dma_attr — DMA attributes structure, 29
ddi_dmae_req — DMA engine request structure, 33

ddi_idevice_cookie — device interrupt cookie, 44

ddi_mapdev_ctl — device mapping-control structure, 45

device interrupt cookie — ddi_idevice_cookie, 44

device mapping-control structure — ddi_mapdev_ctl, 45

device mapping-control structure — devmap_callback_ctl, 46

device operations structure, — dev_ops, 48

devmap_callback_ctl — device mapping-control structure, 46

DMA attributes structure — ddi_dma_attr, 29

DMA cookie structure, — ddi_dma_cookie, 32

DMA engine request structure — ddi_dmae_req, 33

DMA limits structure — ddi_dma_lim, 37, 39

DMA Request structure, — `ddi_dma_req`, 41
driver's message freeing routine, — `free_rtn`, 50
drivers, loadable, linkage structure, —
`modldrv`, 59

F

`fmodsw` — STREAMS module declaration
structure, 49

I

I/O, block, data transfer structure, — `buf`, 17
I/O data storage structure using `uio`, —
`iovec`, 52
I/O request structure, scatter/gather, — `uio`, 92
`iocblk` — STREAMS data structure for the
`M_IOCTL` message type, 51

K

kernel statistics structure — `kstat`, 53
`kstat` — kernel statistics structure, 53
`kstat_intr` — structure for interrupt kstats, 55
`kstat_io` — structure for I/O kstats, 56
`kstat_named` — structure for named kstats, 57

L

`linkblk` — STREAMS data structure sent to
multiplexor drivers to indicate a link, 58

M

`modlinkage` — module linkage structure, 60

O

options structure for `M_SETOPTS` message —
`stroptions`, 87

Q

`queclass` — a STREAMS macro that returns the
queue message class definitions for a given
message block, 66

S

`scsi_address` — SCSI address structure, 68
SCSI address structure — `scsi_address`, 68
`scsi_arq_status` — SCSI auto request sense
structure, 69
SCSI ASC ASCQ to message structure,
`scsi-vu-errmsg`, 70
`scsi_asc_key_strings`, SCSI ASC ASCQ to
message structure, 70
SCSI auto request sense structure —
`scsi_arq_status`, 69
`scsi_device` — SCSI device structure, 71
SCSI device structure — `scsi_device`, 71
SCSI device structure — `scsi_inquiry`, 77
`scsi_extended_sense` — SCSI extended sense
structure, 72
SCSI extended sense structure —
`scsi_extended_sense`, 72
`scsi_hba_tran` — SCSI Host Bus Adapter (HBA)
driver transport vector structure, 75
SCSI Host Bus Adapter (HBA) driver transport
vector structure — `scsi_hba_tran`, 75
`scsi_inquiry` — SCSI device structure, 77
SCSI packet structure — `scsi_pkt`, 80
`scsi_pkt` — SCSI packet structure, 80
`pkt_flags` Definitions, 81
`pkt_reason` Definitions, 82
`pkt_state` Definitions, 83
`pkt_statistics` Definitions, 83
`scsi_status` — SCSI status structure, 84
SCSI status structure — `scsi_status`, 84
STREAMS data structure for the `M_COPYIN`
and the `M_COPYOUT` message types —
`copyreq`, 22
STREAMS data structure for the `M_IOCTLDATA`
message type — `copyresp`, 23
STREAMS data structure for the `M_IOCTL`
message type — `iocblk`, 51
STREAMS data structure sent to multiplexor
drivers to indicate a link — `linkblk`, 58

STREAMS driver identification and limit value structure, — `module_info`, 62

STREAMS entity declaration structure, — `streamtab`, 86

STREAMS macro that returns the queue message class definitions for a given message block — `queclass`, 66

STREAMS message block structure, — `msgb`, 63

STREAMS message data structure, — `datab`, 24

STREAMS module declaration structure — `fmodsw`, 49

STREAMS modules, loadable, linkage structure, `modlstrmod`, 61

STREAMS queue flow control information structure, — `qband`, 64

STREAMS queue processing procedures structure, — `qinit`, 65

STREAMS queue structure, — `queue`, 67

stoptions — options structure for `M_SETOPTS` message, 87

structure for I/O kstats — `kstat_io`, 56

structure for interrupt kstats — `kstat_intr`, 55

structure for named kstats — `kstat_named`, 57

T

tuple — Card Information Structure (CIS) access structure, 89

U

uio — scatter/gather I/O request structure, 92

