SunOS Reference Manual

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Preface

OVERVIEW

A man page is provided for both the naive user, and sophisticated user who is familiar with the SunOS operating system and is in need of on-line information. 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.

The following contains a brief description of each section in the man pages 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 of this volume.

- 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, etc.

- 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 operating systems 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 may 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 pathname is shown. Literal characters (commands and options) are in **bold** font and variables (arguments, parameters and substitution characters) are in *italic* font. 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:

[ ] The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument *must* be specified.

. . . Ellipses. Several values may 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 time.

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file. The protocol specification pathname is always listed in **bold** font.

AVAILABILITY

This section briefly states any limitations on the availabilty of the command. These limitations could be hardware or software specific.

A specification of a class of hardware platform, such as **x86** or **SPARC**, denotes that the command or interface is applicable for the hardware platform specified.

In Section 1 and Section 1M, **AVAILABILITY** indicates which package contains the command being described on the manual page. In order to use the command, the specified package must have been installed with the operating system. If the package was not installed, see **pkgadd**(1) for information on how to upgrade.

MT-LEVEL

This section lists the **MT-LEVEL** of the library functions described in the Section 3 manual pages. The **MT-LEVEL** defines the libraries' ability to support threads. See **Intro**(3) for more information.

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, functions and such, are described under USAGE.

IOCTLS

This section appears on pages in Section 7 only. Only the device class which supplies appropriate parameters to the **ioctls**(2) system call is called **ioctls** and generates its own heading. IOCTLS for a specific device are listed alphabetically (on the man page for that specific device). IOCTLS are used for a particular class of devices all which have an **io** ending, such as **mtio**(7).

OPTIONS

This 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.

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 as **void** do not return values, so they are not discussed in RETURN VALUES.

ERRORS

On failure, most functions place an error code in the global variable **errno** indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE

This section is provided as a *guidance* on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:

Commands
Modifiers
Variables
Expressions
Input Grammar

EXAMPLES

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as

example%

or if the user must be super-user,

example#

Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.

ENVIRONMENT

This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

FILES

This section lists all filenames 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.

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. Messages appear in **bold** font with the exception of variables, which are in *italic* font.

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.

NAME

ASSERT, assert – expression verification

SYNOPSIS

#include <sys/debug.h>

void ASSERT(EX);

ARGUMENTS

EX boolean expression.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

ASSERT ( ) is a macro which checks to see if the expression *EX* is true. If it is not then **ASSERT ( )** causes an error message to be logged to the console and the system to panic. **ASSERT ( )** works only if the preprocessor symbol **DEBUG** is defined.

CONTEXT

ASSERT ( ) can be used from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

Intro, intro – introduction to DDI / DKI functions

DESCRIPTION

Section 9F describes the kernel functions available for use by device drivers.

In this section, the information for each driver function is organized under the following headings:

- **NAME** summarizes the function's purpose.
- **SYNOPSIS** shows the syntax of the function's entry point in the source code. **#include** directives are shown for required headers.
- **INTERFACE LEVEL** describes any architecture dependencies.
- **ARGUMENTS** describes any arguments required to invoke the function.
- **DESCRIPTION** describes general information about the function.
- **RETURN VALUES** describes the return values and messages that can result from invoking the function.
- **CONTEXT** indicates from which driver context (user, kernel, interrupt, or high-level interrupt) the function can be called.

  A driver function has *user context* if it was directly invoked because of a user thread. The **read**(9E) entry point of the driver, invoked by a **read**(2) system call, has user context.

  A driver function has *kernel context* if was invoked by some other part of the kernel. In a block device driver, the **strategy**(9E) entry point may be called by the page daemon to write pages to the device. The page daemon has no relation to the current user thread, so in this case **strategy**(9E) has kernel context.

  Interrupt context is kernel context, but also has an interrupt level associated with it. Driver interrupt routines have interrupt context.

  High-level interrupt context is a more restricted form of interrupt context. If **ddi_intr_hilevel**(9F) indicates that an interrupt is high-level, driver interrupt routines added for that interrupt with **ddi_add_intr**(9F) run in high-level interrupt context. These interrupt routines are only allowed to call **ddi_trigger_softintr**(9F)**, mutex_enter**(9F) and **mutex_exit**(9F). Furthermore, **mutex_enter**(9F) and **mutex_exit**(9F) may only be called on mutexes initialized with the ddi_iblock_cookie returned by **ddi_add_intr**(9F).
- **SEE ALSO** indicates functions that are related by usage and sources, and which can be referred to for further information.
- **EXAMPLES** shows how the function can be used in driver code.

Every driver MUST include **<sys/ddi.h>** and **<sys/sunddi.h>**, in that order, and as the last files the driver includes.

STREAMS Kernel Function Summary

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

Routine	Type
adjmsg	DDI / DKI
allocb	DDI / DKI
backq	DDI / DKI
bcanput	DDI / DKI
bcanputnext	DDI / DKI
bufcall	DDI / DKI
canput	DDI / DKI
canputnext	DDI / DKI
clrbuf	DDI / DKI
copyb	DDI / DKI
copymsg	DDI / DKI
datamsg	DDI / DKI
dupb	DDI / DKI
dupmsg	DDI / DKI
enableok	DDI / DKI
esballoc	DDI / DKI
esbbcall	DDI / DKI
flushband	DDI / DKI
flushq	DDI / DKI
freeb	DDI / DKI
freemsg	DDI / DKI
freezestr	DDI / DKI
getq	DDI / DKI
insq	DDI / DKI
linkb	DDI / DKI
msgdsize	DDI / DKI
msgpullup	DDI / DKI
mt-streams	Solaris DDI
noenable	DDI / DKI
OTHERQ	DDI / DKI
pullupmsg	DDI / DKI
put	DDI / DKI
putbq	DDI / DKI
putctl	DDI / DKI
putctl1	DDI / DKI
putnext	DDI / DKI
putnextctl	DDI / DKI
putq	DDI / DKI
qbufcall	Solaris DDI
qenable	DDI / DKI
qprocson	DDI / DKI

qprocsoff	DDI / DKI
qreply	DDI / DKI
qsize	DDI / DKI
qtimeout	Solaris DDI
qunbufcall	Solaris DDI
quntimeout	Solaris DDI
qwait	Solaris DDI
qwait_sig	Solaris DDI
qwriter	Solaris DDI
RD	DDI / DKI
rmvb	DDI / DKI
rmvq	DDI / DKI
SAMESTR	DDI / DKI
strlog	DDI / DKI
strqget	DDI / DKI
strqset	DDI / DKI
testb	DDI / DKI
unbufcall	DDI / DKI
unfreezestr	DDI / DKI
unlinkb	DDI / DKI
WR	DDI / DKI

The following table summarizes the functions not specific to STREAMS.

Routine	Type
ASSERT	DDI / DKI
bcmp	DDI / DKI
bcopy	DDI / DKI
biodone	DDI / DKI
bioerror	Solaris DDI
bioreset	Solaris DDI
biowait	DDI / DKI
bp_mapin	DDI / DKI
bp_mapout	DDI / DKI
brelse	DDI / DKI
btop	DDI / DKI
btopr	DDI / DKI
bzero	DDI / DKI
cmn_err	DDI / DKI
copyin	DDI / DKI
copyout	DDI / DKI
cv_broadcast	Solaris DDI
cv_destroy	Solaris DDI
cv_init	Solaris DDI
cv_signal	Solaris DDI
cv_timedwait	Solaris DDI
cv_wait	Solaris DDI
cv_wait_sig	Solaris DDI
ddi_add_intr	Solaris DDI
ddi_add_softintr	Solaris DDI
ddi_btop	Solaris DDI
ddi_btopr	Solaris DDI
ddi_copyin	Solaris DDI
ddi_copyout	Solaris DDI
ddi_create_minor_node	Solaris DDI
ddi_dev_is_sid	Solaris DDI
ddi_dev_nintrs	Solaris DDI
ddi_dev_nregs	Solaris DDI
ddi_dev_regsize	Solaris DDI
ddi_dma_addr_setup	Solaris DDI
ddi_dma_buf_setup	Solaris DDI
ddi_dma_burstsizes	Solaris DDI
ddi_dma_coff	Solaris SPARC DDI
ddi_dma_curwin	Solaris SPARC DDI
ddi_dma_devalign	Solaris DDI
ddi_dma_free	Solaris DDI
ddi_dma_htoc	Solaris SPARC DDI

ddi_dma_movwin	Solaris SPARC DDI
ddi_dma_nextseg	Solaris DDI
ddi_dma_nextwin	Solaris DDI
ddi_dma_segtocookie	Solaris DDI
ddi_dma_setup	Solaris DDI
ddi_dma_sync	Solaris DDI
ddi_dmae_alloc	Solaris x86 DDI
ddi_dmae_release	Solaris x86 DDI
ddi_dmae_prog	Solaris x86 DDI
ddi_dmae_disable	Solaris x86 DDI
ddi_dmae_enable	Solaris x86 DDI
ddi_dmae_stop	Solaris x86 DDI
ddi_dmae_getcnt	Solaris x86 DDI
ddi_dmae_1stparty	Solaris x86 DDI
ddi_dmae_getlim	Solaris x86 DDI
ddi_enter_critical	Solaris DDI
ddi_exit_critical	Solaris DDI
ddi_ffs	Solaris DDI
ddi_fls	Solaris DDI
ddi_get_cred	Solaris DDI
ddi_get_driver_private	Solaris DDI
ddi_get_instance	Solaris DDI
ddi_getlongprop	Solaris DDI
ddi_getlongprop_buf	Solaris DDI
ddi_get_name	Solaris DDI
ddi_get_parent	Solaris DDI
ddi_getprop	Solaris DDI
ddi_getproplen	Solaris DDI
ddi_get_soft_state	Solaris DDI
ddi_intr_hilevel	Solaris DDI
ddi_iomin	Solaris DDI
ddi_iopb_alloc	Solaris DDI
ddi_iopb_free	Solaris DDI
ddi_map_regs	Solaris DDI
ddi_mapdev	Solaris DDI
ddi_mapdev_intercept	Solaris DDI
ddi_mapdev_nointercept	Solaris DDI
ddi_mem_alloc	Solaris DDI
ddi_mem_free	Solaris DDI
ddi_peekc	Solaris DDI
ddi_peekd	Solaris DDI
ddi_peekl	Solaris DDI
ddi_peeks	Solaris DDI
ddi_pokec	Solaris DDI
ddi_poked	Solaris DDI

ddi_pokel	Solaris DDI
ddi_pokes	Solaris DDI
ddi_prop_create	Solaris DDI
ddi_prop_modify	Solaris DDI
ddi_prop_op	Solaris DDI
ddi_prop_remove	Solaris DDI
ddi_prop_remove_all	Solaris DDI
ddi_prop_undefine	Solaris DDI
ddi_ptob	Solaris DDI
ddi_remove_intr	Solaris DDI
ddi_remove_minor_node	Solaris DDI
ddi_remove_softintr	Solaris DDI
ddi_report_dev	Solaris DDI
ddi_root_node	Solaris DDI
ddi_segmap	Solaris DDI
ddi_set_driver_private	Solaris DDI
ddi_slaveonly	Solaris DDI
ddi_soft_state	Solaris DDI
ddi_soft_state_fini	Solaris DDI
ddi_soft_state_free	Solaris DDI
ddi_soft_state_init	Solaris DDI
ddi_soft_state_zalloc	Solaris DDI
ddi_trigger_softintr	Solaris DDI
ddi_unmap_regs	Solaris DDI
delay	DDI / DKI
disksort	Solaris DDI
drv_getparm	DDI / DKI
drv_hztousec	DDI / DKI
drv_priv	DDI / DKI
drv_usectohz	DDI / DKI
drv_usecwait	DDI / DKI
free_pktiopb	Solaris DDI
freerbuf	DDI / DKI
geterror	DDI / DKI
getmajor	DDI / DKI
getminor	DDI / DKI
get_pktiopb	Solaris DDI
getrbuf	DDI / DKI
hat_getkpfnum	DKI only
inb	Solaris x86 DDI
inl	Solaris x86 DDI
inw	Solaris x86 DDI
kmem_alloc	DDI / DKI
kmem_free	DDI / DKI
kmem_zalloc	DDI / DKI

kstat_create	Solaris DDI
kstat_delete	Solaris DDI
kstat_install	Solaris DDI
kstat_named_init	Solaris DDI
kstat_queue	Solaris DDI
kstat_runq_enter	Solaris DDI
kstat_runq_exit	Solaris DDI
kstat_runq_back_to_waitq	Solaris DDI
kstat_waitq_enter	Solaris DDI
kstat_waitq_exit	Solaris DDI
kstat_waitq_to_runq	Solaris DDI
makecom_g0	Solaris DDI
makecom_g0_s	Solaris DDI
makecom_g1	Solaris DDI
makecom_g5	Solaris DDI
makedevice	DDI / DKI
max	DDI / DKI
min	DDI / DKI
minphys	Solaris DDI
mod_info	Solaris DDI
mod_install	Solaris DDI
mod_remove	Solaris DDI
mutex_destroy	Solaris DDI
mutex_enter	Solaris DDI
mutex_exit	Solaris DDI
mutex_init	Solaris DDI
mutex_owned	Solaris DDI
mutex_tryenter	Solaris DDI
nochpoll	Solaris DDI
nodev	DDI / DKI
nulldev	DDI / DKI
numtos	Solaris DDI
outb	Solaris x86 DDI
outl	Solaris x86 DDI
outw	Solaris x86 DDI
physio	Solaris DDI
pollwakeup	DDI / DKI
proc_ref	Solaris DDI
proc_signal	Solaris DDI
proc_unref	Solaris DDI
ptob	DDI / DKI
repinsb	Solaris x86 DDI
repinsd	Solaris x86 DDI
repinsw	Solaris x86 DDI
repoutsb	Solaris x86 DDI

repoutsd	Solaris x86 DDI
repoutsw	Solaris x86 DDI
rmalloc	DDI / DKI
rmalloc_wait	DDI / DKI
rmallocmap	DDI / DKI
rmfree	DDI / DKI
rmfreemap	DDI / DKI
rw_destroy	Solaris DDI
rw_downgrade	Solaris DDI
rw_enter	Solaris DDI
rw_exit	Solaris DDI
rw_init	Solaris DDI
rw_read_locked	Solaris DDI
rw_tryenter	Solaris DDI
rw_tryupgrade	Solaris DDI
scsi_abort	Solaris DDI
scsi_alloc_consistent_buf	Solaris DDI
scsi_cname	Solaris DDI
scsi_destroy_pkt	Solaris DDI
scsi_dmafree	Solaris DDI
scsi_dmaget	Solaris DDI
scsi_dname	Solaris DDI
scsi_errmsg	Solaris DDI
scsi_free_consistent_buf	Solaris DDI
scsi_hba_attach	Solaris DDI
scsi_hba_detach	Solaris DDI
scsi_hba_fini	Solaris DDI
scsi_hba_init	Solaris DDI
scsi_hba_lookup_capstr	Solaris DDI
scsi_hba_pkt_alloc	Solaris DDI
scsi_hba_pkt_free	Solaris DDI
scsi_hba_probe	Solaris DDI
scsi_hba_tran_alloc	Solaris DDI
scsi_hba_tran_free	Solaris DDI
scsi_ifgetcap	Solaris DDI
scsi_ifsetcap	Solaris DDI
scsi_init_pkt	Solaris DDI
scsi_log	Solaris DDI
scsi_mname	Solaris DDI
scsi_pktalloc	Solaris DDI
scsi_pktfree	Solaris DDI
scsi_poll	Solaris DDI
scsi_probe	Solaris DDI
scsi_resalloc	Solaris DDI
scsi_reset	Solaris DDI

scsi_resfree	Solaris DDI
scsi_rname	Solaris DDI
scsi_slave	Solaris DDI
scsi_sname	Solaris DDI
scsi_sync_pkt	Solaris DDI
scsi_transport	Solaris DDI
scsi_unprobe	Solaris DDI
scsi_unslave	Solaris DDI
sema_destroy	Solaris DDI
sema_init	Solaris DDI
sema_p	Solaris DDI
sema_p_sig	Solaris DDI
sema_tryp	Solaris DDI
sema_v	Solaris DDI
sprintf	Solaris DDI
stoi	Solaris DDI
strchr	Solaris DDI
strcmp	Solaris DDI
strcpy	Solaris DDI
strlen	Solaris DDI
strncmp	Solaris DDI
strncpy	Solaris DDI
swab	DDI / DKI
timeout	DDI / DKI
uiomove	DDI / DKI
untimeout	DDI / DKI
ureadc	DDI / DKI
uwritec	DDI / DKI
vcmn_err	DDI / DKI
vsprintf	Solaris DDI

Name	**Appears on Page**	**Description**
adjmsg	**adjmsg**(9F)	trim bytes from a message
allocb	**allocb**(9F)	allocate a message block
ASSERT	**ASSERT**(9F)	expression verification
assert	**ASSERT**(9F)	expression verification
backq	**backq**(9F)	get pointer to the queue behind the current queue

bcanput	**bcanput**(9F)	test for flow control in specified priority band
bcanputnext	**canputnext**(9F)	test for room in next module’s message queue
bcmp	**bcmp**(9F)	compare two byte arrays
bcopy	**bcopy**(9F)	copy data between address locations in the kernel
biodone	**biodone**(9F)	release buffer after buffer I/O transfer and notify blocked threads
bioerror	**bioerror**(9F)	indicate error in buffer header
bioreset	**bioreset**(9F)	reuse a private buffer header after I/O is complete
biowait	**biowait**(9F)	suspend processes pending completion of block I/O
bp_mapin	**bp_mapin**(9F)	allocate virtual address space
bp_mapout	**bp_mapout**(9F)	deallocate virtual address space
brelse	**brelse**(9F)	return buffer to the free list
btop	**btop**(9F)	convert size in bytes to size in pages (round down)
btopr	**btopr**(9F)	convert size in bytes to size in pages (round up)
bufcall	**bufcall**(9F)	call a function when a buffer becomes available
bzero	**bzero**(9F)	clear memory for a given number of bytes
canput	**canput**(9F)	test for room in a message queue
canputnext	**canputnext**(9F)	test for room in next module’s message queue
clrbuf	**clrbuf**(9F)	erase the contents of a buffer

cmn_err	**cmn_err**(9F)	display an error message or panic the system
condvar	**condvar**(9F)	condition variable routines
copyb	**copyb**(9F)	copy a message block
copyin	**copyin**(9F)	copy data from a user program to a driver buffer
copymsg	**copymsg**(9F)	copy a message
copyout	**copyout**(9F)	copy data from a driver to a user program
cv_broadcast	**condvar**(9F)	condition variable routines
cv_destroy	**condvar**(9F)	condition variable routines
cv_init	**condvar**(9F)	condition variable routines
cv_signal	**condvar**(9F)	condition variable routines
cv_timedwait	**condvar**(9F)	condition variable routines
cv_timedwait_sig	**condvar**(9F)	condition variable routines
cv_wait	**condvar**(9F)	condition variable routines
cv_wait_sig	**condvar**(9F)	condition variable routines
datamsg	**datamsg**(9F)	test whether a message is a data message
ddi_add_intr	**ddi_add_intr**(9F)	add and remove an interrupt handler
ddi_add_softintr	**ddi_add_softintr**(9F)	add, remove or trigger a soft interrupt
ddi_btop	**ddi_btop**(9F)	page size conversions
ddi_btopr	**ddi_btop**(9F)	page size conversions
ddi_copyin	**ddi_copyin**(9F)	copy data to a driver buffer
ddi_copyout	**ddi_copyout**(9F)	copy data from a driver
ddi_create_minor_node	**ddi_create_minor_node**(9F)	create a minor node for this device
ddi_dev_is_sid	**ddi_dev_is_sid**(9F)	tell whether a device is self-identifying

ddi_dev_nintrs	**ddi_dev_nintrs**(9F)	return the number of interrupt specifications a device has
ddi_dev_nregs	**ddi_dev_nregs**(9F)	return the number of register sets a device has
ddi_dev_regsize	**ddi_dev_regsize**(9F)	return the size of a device's register
ddi_dma_addr_setup	**ddi_dma_addr_setup**(9F)	easier DMA setup for use with virtual addresses
ddi_dma_buf_setup	**ddi_dma_buf_setup**(9F)	easier DMA setup for use with buffer structures
ddi_dma_burstsizes	**ddi_dma_burstsizes**(9F)	find out the allowed burst sizes for a DMA mapping
ddi_dma_coff	**ddi_dma_coff**(9F)	convert a DMA cookie to an offset within a DMA handle
ddi_dma_curwin	**ddi_dma_curwin**(9F)	report current DMA window offset and size
ddi_dma_devalign	**ddi_dma_devalign**(9F)	find DMA mapping alignment and minimum transfer size
ddi_dma_free	**ddi_dma_free**(9F)	release system DMA resources
ddi_dma_htoc	**ddi_dma_htoc**(9F)	convert a DMA handle to a DMA address cookie
ddi_dma_movwin	**ddi_dma_movwin**(9F)	shift current DMA window
ddi_dma_nextseg	**ddi_dma_nextseg**(9F)	get next DMA segment
ddi_dma_nextwin	**ddi_dma_nextwin**(9F)	get next DMA window
ddi_dma_segtocookie	**ddi_dma_segtocookie**(9F)	convert a DMA segment to a DMA address cookie
ddi_dma_setup	**ddi_dma_setup**(9F)	setup DMA resources
ddi_dma_sync	**ddi_dma_sync**(9F)	synchronize CPU and I/O views of memory
ddi_dmae	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_1stparty	**ddi_dmae**(9F)	system DMA engine functions

ddi_dmae_alloc	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_disable	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_enable	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_getcnt	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_getlim	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_prog	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_release	**ddi_dmae**(9F)	system DMA engine functions
ddi_dmae_stop	**ddi_dmae**(9F)	system DMA engine functions
ddi_enter_critical	**ddi_enter_critical**(9F)	enter and exit a critical region of control
ddi_exit_critical	**ddi_enter_critical**(9F)	enter and exit a critical region of control
ddi_ffs	**ddi_ffs**(9F)	find first (last) bit set in a long integer
ddi_fls	**ddi_ffs**(9F)	find first (last) bit set in a long integer
ddi_get_cred	**ddi_get_cred**(9F)	returns a pointer to the credential structure of the caller.
ddi_get_driver_private	**ddi_get_driver_private**(9F)	get or set the address of the device’s private data area
ddi_get_instance	**ddi_get_instance**(9F)	get device instance number
ddi_get_name	**ddi_get_name**(9F)	return the devinfo node name
ddi_get_parent	**ddi_get_parent**(9F)	find the parent of a device information structure
ddi_get_soft_state	**ddi_soft_state**(9F)	driver soft state utility routines

ddi_getlongprop	**ddi_prop_op**(9F)	get property information for leaf device drivers
ddi_getlongprop_buf	**ddi_prop_op**(9F)	get property information for leaf device drivers
ddi_getprop	**ddi_prop_op**(9F)	get property information for leaf device drivers
ddi_getproplen	**ddi_prop_op**(9F)	get property information for leaf device drivers
ddi_intr_hilevel	**ddi_intr_hilevel**(9F)	indicate interrupt handler type
ddi_iomin	**ddi_iomin**(9F)	find minimum alignment and transfer size for DMA
ddi_iopb_alloc	**ddi_iopb_alloc**(9F)	allocate and free non-sequentially accessed memory
ddi_iopb_free	**ddi_iopb_alloc**(9F)	allocate and free non-sequentially accessed memory
ddi_map_regs	**ddi_map_regs**(9F)	map or unmap registers
ddi_mapdev	**ddi_mapdev**(9F)	create driver-controlled mapping of device
ddi_mapdev_intercept	**ddi_mapdev_intercept**(9F)	control driver notification of user accesses
ddi_mapdev_nointercept	**ddi_mapdev_intercept**(9F)	control driver notification of user accesses
ddi_mem_alloc	**ddi_mem_alloc**(9F)	allocate and free sequentially accessed memory
ddi_mem_free	**ddi_mem_alloc**(9F)	allocate and free sequentially accessed memory
ddi_peek	**ddi_peek**(9F)	read a value from a location
ddi_peekc	**ddi_peek**(9F)	read a value from a location

ddi_peekd	**ddi_peek**(9F)	read a value from a location
ddi_peekl	**ddi_peek**(9F)	read a value from a location
ddi_peeks	**ddi_peek**(9F)	read a value from a location
ddi_poke	**ddi_poke**(9F)	write a value to a location
ddi_pokec	**ddi_poke**(9F)	write a value to a location
ddi_poked	**ddi_poke**(9F)	write a value to a location
ddi_pokel	**ddi_poke**(9F)	write a value to a location
ddi_pokes	**ddi_poke**(9F)	write a value to a location
ddi_prop_create	**ddi_prop_create**(9F)	create, remove, or modify properties for leaf device drivers
ddi_prop_modify	**ddi_prop_create**(9F)	create, remove, or modify properties for leaf device drivers
ddi_prop_op	**ddi_prop_op**(9F)	get property information for leaf device drivers
ddi_prop_remove	**ddi_prop_create**(9F)	create, remove, or modify properties for leaf device drivers
ddi_prop_remove_all	**ddi_prop_create**(9F)	create, remove, or modify properties for leaf device drivers
ddi_prop_undefine	**ddi_prop_create**(9F)	create, remove, or modify properties for leaf device drivers
ddi_ptob	**ddi_btop**(9F)	page size conversions
ddi_remove_intr	**ddi_add_intr**(9F)	add and remove an interrupt handler
ddi_remove_minor_node	**ddi_remove_minor_node**(9F)	remove a minor node for this dev_info
ddi_remove_softintr	**ddi_add_softintr**(9F)	add, remove or trigger a soft interrupt
ddi_report_dev	**ddi_report_dev**(9F)	announce a device
ddi_root_node	**ddi_root_node**(9F)	get the root of the dev_info tree
ddi_segmap	**ddi_segmap**(9F)	map a segment

ddi_set_driver_private	**ddi_get_driver_private**(9F)	get or set the address of the device's private data area
ddi_slaveonly	**ddi_slaveonly**(9F)	tell if a device is installed in a slave access only location
ddi_soft_state	**ddi_soft_state**(9F)	driver soft state utility routines
ddi_soft_state_fini	**ddi_soft_state**(9F)	driver soft state utility routines
ddi_soft_state_free	**ddi_soft_state**(9F)	driver soft state utility routines
ddi_soft_state_init	**ddi_soft_state**(9F)	driver soft state utility routines
ddi_soft_state_zalloc	**ddi_soft_state**(9F)	driver soft state utility routines
ddi_trigger_softintr	**ddi_add_softintr**(9F)	add, remove or trigger a soft interrupt
ddi_unmap_regs	**ddi_map_regs**(9F)	map or unmap registers
delay	**delay**(9F)	delay execution for a specified number of clock ticks
disksort	**disksort**(9F)	single direction elevator seek sort for buffers
drv_getparm	**drv_getparm**(9F)	retrieve kernel state information
drv_hztousec	**drv_hztousec**(9F)	convert clock ticks to microseconds
drv_priv	**drv_priv**(9F)	determine driver privilege
drv_usectohz	**drv_usectohz**(9F)	convert microseconds to clock ticks
drv_usecwait	**drv_usecwait**(9F)	busy-wait for specified interval
dupb	**dupb**(9F)	duplicate a message block descriptor
dupmsg	**dupmsg**(9F)	duplicate a message
enableok	**enableok**(9F)	reschedule a queue for service
esballoc	**esballoc**(9F)	allocate a message block using a caller-supplied buffer

esbbcall	**esbbcall**(9F)	call function when buffer is available
flushband	**flushband**(9F)	flush messages for a specified priority band
flushq	**flushq**(9F)	remove messages from a queue
free_pktiopb	**get_pktiopb**(9F)	allocate / free a SCSI packet in the iopb map
freeb	**freeb**(9F)	free a message block
freemsg	**freemsg**(9F)	free all message blocks in a message
freerbuf	**freerbuf**(9F)	free a raw buffer header
freezestr	**freezestr**(9F)	freeze, thaw the state of a stream
get_pktiopb	**get_pktiopb**(9F)	allocate / free a SCSI packet in the iopb map
geterror	**geterror**(9F)	return I / O error
getmajor	**getmajor**(9F)	get major device number
getminor	**getminor**(9F)	get minor device number
getq	**getq**(9F)	get the next message from a queue
getrbuf	**getrbuf**(9F)	get a raw buffer header
hat_getkpfnum	**hat_getkpfnum**(9F)	get page frame number for kernel address
inb	**inb**(9F)	read from an I / O port
inl	**inb**(9F)	read from an I / O port
insq	**insq**(9F)	insert a message into a queue
inw	**inb**(9F)	read from an I / O port
kmem_alloc	**kmem_alloc**(9F)	allocate space from kernel free memory
kmem_free	**kmem_free**(9F)	free previously allocated kernel memory
kmem_zalloc	**kmem_zalloc**(9F)	allocate and clear space from kernel free memory
kstat_create	**kstat_create**(9F)	create and initialize a new kstat
kstat_delete	**kstat_delete**(9F)	remove a kstat from the system
kstat_install	**kstat_install**(9F)	add a fully initialized kstat to the system

kstat_named_init	**kstat_named_init**(9F)	initialize a named kstat
kstat_queue	**kstat_queue**(9F)	update I/O kstat statistics
kstat_runq_back_to_waitq	**kstat_queue**(9F)	update I/O kstat statistics
kstat_runq_enter	**kstat_queue**(9F)	update I/O kstat statistics
kstat_runq_exit	**kstat_queue**(9F)	update I/O kstat statistics
kstat_waitq_enter	**kstat_queue**(9F)	update I/O kstat statistics
kstat_waitq_exit	**kstat_queue**(9F)	update I/O kstat statistics
kstat_waitq_to_runq	**kstat_queue**(9F)	update I/O kstat statistics
linkb	**linkb**(9F)	concatenate two message blocks
makecom	**makecom**(9F)	make a packet for SCSI commands
makecom_g0	**makecom**(9F)	make a packet for SCSI commands
makecom_g0_s	**makecom**(9F)	make a packet for SCSI commands
makecom_g1	**makecom**(9F)	make a packet for SCSI commands
makecom_g5	**makecom**(9F)	make a packet for SCSI commands
makedevice	**makedevice**(9F)	make device number from major and minor numbers
max	**max**(9F)	return the larger of two integers
min	**min**(9F)	return the lesser of two integers
minphys	**physio**(9F)	perform physical I/O
mod_info	**mod_install**(9F)	add, remove or query a loadable module
mod_install	**mod_install**(9F)	add, remove or query a loadable module
mod_remove	**mod_install**(9F)	add, remove or query a loadable module
msgdsize	**msgdsize**(9F)	return the number of bytes in a message

msgpullup	**msgpullup**(9F)	concatenate bytes in a message
mt-streams	**mt-streams**(9F)	STREAMS multithreading
mutex	**mutex**(9F)	mutual exclusion lock routines
mutex_destroy	**mutex**(9F)	mutual exclusion lock routines
mutex_enter	**mutex**(9F)	mutual exclusion lock routines
mutex_exit	**mutex**(9F)	mutual exclusion lock routines
mutex_init	**mutex**(9F)	mutual exclusion lock routines
mutex_owned	**mutex**(9F)	mutual exclusion lock routines
mutex_tryenter	**mutex**(9F)	mutual exclusion lock routines
nochpoll	**nochpoll**(9F)	error return function for non-pollable devices.
nodev	**nodev**(9F)	error return function
noenable	**noenable**(9F)	prevent a queue from being scheduled
nulldev	**nulldev**(9F)	zero return function
numtos	**stoi**(9F)	convert between an integer and a decimal string
OTHERQ	**OTHERQ**(9F)	get pointer to queue's partner queue
otherq	**OTHERQ**(9F)	get pointer to queue's partner queue
outb	**outb**(9F)	write to an I/O port
outl	**outb**(9F)	write to an I/O port
outw	**outb**(9F)	write to an I/O port
physio	**physio**(9F)	perform physical I/O
pollwakeup	**pollwakeup**(9F)	inform a process that an event has occurred
proc_ref	**proc_signal**(9F)	send a signal to a process
proc_signal	**proc_signal**(9F)	send a signal to a process
proc_unref	**proc_signal**(9F)	send a signal to a process

ptob	**ptob**(9F)	convert size in pages to size in bytes
pullupmsg	**pullupmsg**(9F)	concatenate bytes in a message
put	**put**(9F)	call a STREAMS put procedure
putbq	**putbq**(9F)	place a message at the head of a queue
putctl1	**putctl1**(9F)	send a control message with a one-byte parameter to a queue
putctl	**putctl**(9F)	send a control message to a queue
putnext	**putnext**(9F)	send a message to the next queue
putnextctl1	**putnextctl1**(9F)	send a control message with a one-byte parameter to a queue
putnextctl	**putnextctl**(9F)	send a control message to a queue
putq	**putq**(9F)	put a message on a queue
qbufcall	**qbufcall**(9F)	call a function when a buffer becomes available
qenable	**qenable**(9F)	enable a queue
qprocsoff	**qprocson**(9F)	enable, disable put and service routines
qprocson	**qprocson**(9F)	enable, disable put and service routines
qreply	**qreply**(9F)	send a message on a stream in the reverse direction
qsize	**qsize**(9F)	find the number of messages on a queue
qtimeout	**qtimeout**(9F)	execute a function after a specified length of time
qunbufcall	**qunbufcall**(9F)	cancel a pending qbufcall request
quntimeout	**quntimeout**(9F)	cancel previous qtimeout function call
qwait	**qwait**(9F)	STREAMS wait routines
qwait_sig	**qwait**(9F)	STREAMS wait routines

qwriter	**qwriter**(9F)	asynchronous STREAMS perimeter upgrade
RD	**RD**(9F)	get pointer to the read queue
rd	**RD**(9F)	get pointer to the read queue
repinsb	**inb**(9F)	read from an I/O port
repinsd	**inb**(9F)	read from an I/O port
repinsw	**inb**(9F)	read from an I/O port
repoutsb	**outb**(9F)	write to an I/O port
repoutsd	**outb**(9F)	write to an I/O port
repoutsw	**outb**(9F)	write to an I/O port
rmalloc	**rmalloc**(9F)	allocate space from a resource map
rmalloc_wait	**rmalloc_wait**(9F)	allocate space from a resource map, wait if necessary
rmallocmap	**rmallocmap**(9F)	allocate and free (respectively) resource maps
rmfree	**rmfree**(9F)	free space back into a resource map
rmfreemap	**rmallocmap**(9F)	allocate and free (respectively) resource maps
rmvb	**rmvb**(9F)	remove a message block from a message
rmvq	**rmvq**(9F)	remove a message from a queue
rw_destroy	**rwlock**(9F)	readers/writer lock functions
rw_downgrade	**rwlock**(9F)	readers/writer lock functions
rw_enter	**rwlock**(9F)	readers/writer lock functions
rw_exit	**rwlock**(9F)	readers/writer lock functions
rw_init	**rwlock**(9F)	readers/writer lock functions
rw_read_locked	**rwlock**(9F)	readers/writer lock functions
rw_tryenter	**rwlock**(9F)	readers/writer lock functions

rw_tryupgrade	**rwlock**(9F)	readers / writer lock functions
rwlock	**rwlock**(9F)	readers / writer lock functions
SAMESTR	**SAMESTR**(9F)	test if next queue is in the same stream
samestr	**SAMESTR**(9F)	test if next queue is in the same stream
scsi_abort	**scsi_abort**(9F)	abort a SCSI command
scsi_alloc_consistent_buf	**scsi_alloc_consistent_buf**(9F)	allocate an I / O buffer for SCSI DMA
scsi_cname	**scsi_cname**(9F)	decode a SCSI name
scsi_destroy_pkt	**scsi_destroy_pkt**(9F)	free an allocated SCSI packet and its DMA resource
scsi_dmafree	**scsi_dmaget**(9F)	SCSI dma utility routines
scsi_dmaget	**scsi_dmaget**(9F)	SCSI dma utility routines
scsi_dname	**scsi_cname**(9F)	decode a SCSI name
scsi_errmsg	**scsi_errmsg**(9F)	display a SCSI request sense message
scsi_free_consistent_buf	**scsi_free_consistent_buf**(9F)	free a previously allocated SCSI DMA I / O buffer
scsi_hba_attach	**scsi_hba_attach**(9F)	SCSI HBA attach and detach routines
scsi_hba_detach	**scsi_hba_attach**(9F)	SCSI HBA attach and detach routines
scsi_hba_fini	**scsi_hba_init**(9F)	SCSI Host Bus Adapter system initialization and completion routines
scsi_hba_init	**scsi_hba_init**(9F)	SCSI Host Bus Adapter system initialization and completion routines
scsi_hba_lookup_capstr	**scsi_hba_lookup_capstr**(9F)	return index matching capability string
scsi_hba_pkt_alloc	**scsi_hba_pkt_alloc**(9F)	allocate and free a scsi_pkt structure
scsi_hba_pkt_free	**scsi_hba_pkt_alloc**(9F)	allocate and free a scsi_pkt structure
scsi_hba_probe	**scsi_hba_probe**(9F)	default SCSI HBA probe function

scsi_hba_tran_alloc	**scsi_hba_tran_alloc**(9F)	allocate and free transport structures
scsi_hba_tran_free	**scsi_hba_tran_alloc**(9F)	allocate and free transport structures
scsi_ifgetcap	**scsi_ifgetcap**(9F)	get / set SCSI transport capability
scsi_ifsetcap	**scsi_ifgetcap**(9F)	get / set SCSI transport capability
scsi_init_pkt	**scsi_init_pkt**(9F)	prepare a complete SCSI packet
scsi_log	**scsi_log**(9F)	display a SCSI-device-related message
scsi_mname	**scsi_cname**(9F)	decode a SCSI name
scsi_pktalloc	**scsi_pktalloc**(9F)	SCSI packet utility routines
scsi_pktfree	**scsi_pktalloc**(9F)	SCSI packet utility routines
scsi_poll	**scsi_poll**(9F)	run a polled SCSI command on behalf of a target driver
scsi_probe	**scsi_probe**(9F)	utility for probing a scsi device
scsi_resalloc	**scsi_pktalloc**(9F)	SCSI packet utility routines
scsi_reset	**scsi_reset**(9F)	reset a SCSI bus or target
scsi_resfree	**scsi_pktalloc**(9F)	SCSI packet utility routines
scsi_rname	**scsi_cname**(9F)	decode a SCSI name
scsi_slave	**scsi_slave**(9F)	utility for SCSI target drivers to establish the presence of a target
scsi_sname	**scsi_cname**(9F)	decode a SCSI name
scsi_sync_pkt	**scsi_sync_pkt**(9F)	synchronize CPU and I / O views of memory
scsi_transport	**scsi_transport**(9F)	request by a SCSI target driver to start a command
scsi_unprobe	**scsi_unprobe**(9F)	free resources allocated during initial probing

scsi_unslave	**scsi_unprobe**(9F)	free resources allocated during initial probing
sema_destroy	**semaphore**(9F)	semaphore functions
sema_init	**semaphore**(9F)	semaphore functions
sema_p	**semaphore**(9F)	semaphore functions
sema_p_sig	**semaphore**(9F)	semaphore functions
sema_tryp	**semaphore**(9F)	semaphore functions
sema_v	**semaphore**(9F)	semaphore functions
semaphore	**semaphore**(9F)	semaphore functions
sprintf	**sprintf**(9F)	format characters in memory
stoi	**stoi**(9F)	convert between an integer and a decimal string
strchr	**strchr**(9F)	find a character in a string
strcmp	**strcmp**(9F)	compare two null terminated strings.
strcpy	**strcpy**(9F)	copy a string from one location to another.
strlen	**strlen**(9F)	determine the number of non-null bytes in a string.
strlog	**strlog**(9F)	submit messages to the log driver
strncmp	**strcmp**(9F)	compare two null terminated strings.
strncpy	**strcpy**(9F)	copy a string from one location to another.
strqget	**strqget**(9F)	get information about a queue or band of the queue
strqset	**strqset**(9F)	change information about a queue or band of the queue
swab	**swab**(9F)	swap bytes in 16-bit halfwords
testb	**testb**(9F)	check for an available buffer
timeout	**timeout**(9F)	execute a function after a specified length of time

uiomove	**uiomove**(9F)	copy kernel data using uio structure
unbufcall	**unbufcall**(9F)	cancel a pending bufcall request
unfreezestr	**freezestr**(9F)	freeze, thaw the state of a stream
unlinkb	**unlinkb**(9F)	remove a message block from the head of a message
untimeout	**untimeout**(9F)	cancel previous timeout function call
ureadc	**ureadc**(9F)	add character to a uio structure
uwritec	**uwritec**(9F)	remove a character from a uio structure
vcmn_err	**cmn_err**(9F)	display an error message or panic the system
vsprintf	**vsprintf**(9F)	format characters in memory
WR	**WR**(9F)	get pointer to the write queue for this module or driver
wr	**WR**(9F)	get pointer to the write queue for this module or driver

NAME

OTHERQ, otherq – get pointer to queue's partner queue

SYNOPSIS

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

queue_t *OTHERQ(queue_t **q***);**

ARGUMENTS

q Pointer to the queue.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

The **OTHERQ( )** function returns a pointer to the other of the two **queue( )** structures that make up a STREAMS module or driver. If *q* points to the read queue the write queue will be returned, and vice versa.

RETURN VALUES

OTHERQ returns a pointer to a queue's partner.

CONTEXT

OTHERQ( ) can be called from user or interrupt context.

EXAMPLES

This routine sets the minimum packet size, the maximum packet size, the high water mark, and the low water mark for the read and write queues of a given module or driver. It is passed either one of the queues. This could be used if a module or driver wished to update its queue parameters dynamically.

```
void
set_q_params(q, min, max, hi, lo)
    queue_t *q;
    short min;
    short max;
    ushort hi;
    ushort lo;
{
        q->q_minpsz = min;
        q->q_maxpsz = max;
        q->q_hiwat = hi;
        q->q_lowat = lo;
        OTHERQ(q)->q_minpsz = min;
        OTHERQ(q)->q_maxpsz = max;
        OTHERQ(q)->q_hiwat = hi;
        OTHERQ(q)->q_lowat = lo;
}
```

SEE ALSO

Writing Device Drivers
STREAMS Programmer's Guide

NAME

RD, rd – get pointer to the read queue

SYNOPSIS

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

queue_t ∗**RD(queue_t** ∗*q***);**

ARGUMENTS

q Pointer to the *write* queue whose *read* queue is to be returned.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

The **RD**( ) function accepts a *write* queue pointer as an argument and returns a pointer to the *read* queue of the same module.

CAUTION: Make sure the argument to this function is a pointer to a *write* queue. **RD**( ) will not check for queue type, and a system panic could result if it is not the right type.

RETURN VALUES

The pointer to the *read* queue.

CONTEXT

RD( ) can be called from user or interrupt context.

EXAMPLES

See the **qreply**(9F) function page for an example of **RD**( ).

SEE ALSO

WR(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

SAMESTR, samestr – test if next queue is in the same stream

SYNOPSIS

#include <sys/stream.h>

int SAMESTR(queue_t **q***);**

ARGUMENTS

q Pointer to the queue.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

The **SAMESTR**( ) function is used to see if the next queue in a stream (if it exists) is the same type as the current queue (that is, both are read queues or both are write queues). This function accounts for the twisted queue connections that occur in a STREAMS pipe and should be used in preference to direct examination of the **q_next** field of **queue**(9S) to see if the stream continues beyond *q*.

RETURN VALUES

SAMESTR( ) returns **1** if the next queue is the same type as the current queue. It returns **0** if the next queue does not exist or if it is not the same type.

CONTEXT

SAMESTR( ) can be called from user or interrupt context.

SEE ALSO

OTHERQ(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

WR, wr – get pointer to the write queue for this module or driver

SYNOPSIS

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

queue_t *WR(queue_t *q);
```

ARGUMENTS

q Pointer to the *read* queue whose *write* queue is to be returned.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

The **WR( )** function accepts a *read* queue pointer as an argument and returns a pointer to the *write* queue of the same module.

CAUTION: Make sure the argument to this function is a pointer to a *read* queue. **WR( )** will not check for queue type, and a system panic could result if the pointer is not to a *read* queue.

RETURN VALUES

The pointer to the *write* queue.

CONTEXT

WR( ) can be called from user or interrupt context.

EXAMPLES

In a STREAMS **close** (9E) routine, the driver or module is passed a pointer to the read queue. These usually are set to the address of the module-specific data structure for the minor device.

```
xxxclose(q, flag)
    queue_t *q;
    int flag;
{
        q->q_ptr = NULL;
        WR(q)->q_ptr = NULL;
        . . .
}
```

SEE ALSO

OTHERQ(9F), **RD**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

adjmsg – trim bytes from a message

SYNOPSIS

#include <sys/stream.h>

int adjmsg(mblk_t *∗mp*, **int** *len***);**

ARGUMENTS

mp Pointer to the message to be trimmed.

len The number of bytes to be removed.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

adjmsg( ) removes bytes from a message. | *len* | (the absolute value of *len*) specifies the number of bytes to be removed. If *len* is greater than **0**, **adjmsg**( ) removes bytes from the head of the message. If *len* is less than **0**, it removes bytes from the tail. **adjmsg**( ) fails if | *len* | is greater than the number of bytes in the message.

RETURN VALUES

adjmsg( ) returns:

1 on success.

0 on failure.

CONTEXT

adjmsg( ) can be called from user or interrupt context.

SEE ALSO

STREAMS Programmer's Guide

NAME

allocb – allocate a message block

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**allocb(int** *size*, **uint** *pri***);**

ARGUMENTS

size The number of bytes in the message block.

pri Priority of the request (no longer used).

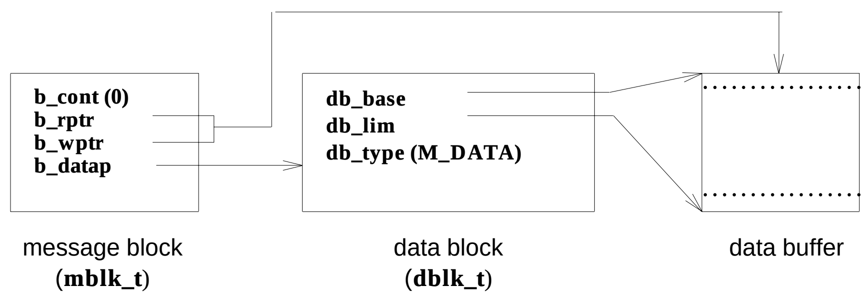
INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

allocb( ) tries to allocate a STREAMS message block. Buffer allocation fails only when the system is out of memory. If no buffer is available, the **bufcall**(9F) function can help a module recover from an allocation failure.

The following figure identifies the data structure members that are affected when a message block is allocated.



RETURN VALUES

A pointer to the allocated message block of type **M_DATA** on success.

A **NULL** pointer on failure.

CONTEXT

allocb( ) can be called from user or interrupt context.

EXAMPLE

Given a pointer to a queue (*q*) and an error number (*err*), the **send_error( )** routine sends an **M_ERROR** type message to the stream head.

If a message cannot be allocated, **NULL** is returned, indicating an allocation failure (line 8). Otherwise, the message type is set to **M_ERROR** (line 10). Line 11 increments the write pointer (**bp->b_wptr**) by the size (one byte) of the data in the message.

A message must be sent up the read side of the stream to arrive at the stream head. To determine whether *q* points to a read queue or to a write queue, the **q->q_flag** member is tested to see if **QREADR** is set (line 13). If it is not set, *q* points to a write queue, and in line 14 the **RD**(9F) function is used to find the corresponding read queue. In line 15, the **putnext**(9F) function is used to send the message upstream, returning **1** if successful.

```
send_error(q,err)
        queue_t *q;
        unsigned char err;
{
        mblk_t *bp;

        if ((bp = allocb(1, BPRI_HI)) == NULL)    /* allocate msg. block */
                return(0);

        bp->b_datap->db_type = M_ERROR;           /* set msg type to M_ERROR */
        *bp->b_wptr++ = err;                      /* increment write pointer */

        if (!(q->q_flag & QREADR))                /* if not read queue    */
                q = RD(q);                        /*    get read queue    */
        putnext(q,bp);                            /* send message upstream */
        return(1);
}
```

SEE ALSO **bufcall**(9F), **esballoc**(9F), **esbbcall**(9F), **testb**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES The *pri* argument is no longer used, but is retained for compatibility with existing drivers.

NAME

backq – get pointer to the queue behind the current queue

SYNOPSIS

#include <sys/stream.h>

queue_t ∗**backq(queue_t** ∗*cq***);**

ARGUMENTS

cq The pointer to the current queue. **queue_t** is an alias for the **queue**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

backq( ) returns a pointer to the queue preceding *cq* (the current queue). If *cq* is a read queue, **backq( )** returns a pointer to the queue downstream from *cq*, unless it is the stream end. If *cq* is a write queue, **backq( )** returns a pointer to the next queue upstream from *cq*, unless it is the stream head.

RETURN VALUES

If successful, **backq( )** returns a pointer to the queue preceding the current queue. Otherwise, it returns NULL.

CONTEXT

backq( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers
STREAMS Programmer's Guide

NAME

bcanput – test for flow control in specified priority band

SYNOPSIS

#include <sys/stream.h>

int bcanput(queue_t **q*, **unsigned char** *pri***);**

ARGUMENTS

q Pointer to the message queue.

pri Message priority.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

bcanput( ) searches through the stream (starting at *q*) until it finds a queue containing a service routine where the message can be enqueued, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if there is room for a message of priority *pri* in the queue.

If *pri* is **0**, **bcanput**( ) is equivalent to a call with **canput**(9F).

canputnext(*q***)** and **bcanputnext(***q***,** *pri***)** should always be used in preference to **canput(***q*→**q_next)** and **bcanput(***q*→**q_next,** *pri***)** respectively.

RETURN VALUES

1 If a message of priority *pri* can be placed on the queue.

0 If the priority band is full.

CONTEXT

bcanput( ) can be called from user or interrupt context.

WARNINGS

Drivers are responsible for both testing a queue with **bcanput**( ) and refraining from placing a message on the queue if **bcanput**( ) fails.

SEE ALSO

bcanputnext(9F), **canput**(9F), **canputnext**(9F), **putbq**(9F), **putnext**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

bcmp – compare two byte arrays

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

int bcmp(char **s1*, **char** ** s2*, **size_t** *len***);**

ARGUMENTS

s1	Pointer to the first character string.
s2	Pointer to the second character string.
len	Number of bytes to be compared.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

bcmp( ) compares two byte arrays of length *len*.

RETURN VALUES

bcmp( ) returns 0 if the arrays are identical, or 1 if they are not.

CONTEXT

bcmp( ) can be called from user or interrupt context.

SEE ALSO

strcmp(9F)

Writing Device Drivers

NOTES

Unlike **strcmp**(9F), **bcmp**( ) does not terminate when it encounters a null byte.

NAME

bcopy – copy data between address locations in the kernel

SYNOPSIS

#include <sys/types.h>

void bcopy(caddr_t *from***, caddr_t** *to***, size_t** *bcount***);**

ARGUMENTS

from Source address from which the copy is made.

to Destination address to which copy is made.

bcount The number of bytes moved.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

bcopy( ) copies *bcount* bytes from one kernel address to another. If the input and output addresses overlap, the command executes, but the results may not be as expected.

Note that **bcopy**( ) should never be used to move data in or out of a user buffer, because it has no provision for handling page faults. The user address space can be swapped out at any time, and **bcopy**( ) always assumes that there will be no paging faults. If **bcopy**( ) attempts to access the user buffer when it is swapped out, the system will panic. It is safe to use **bcopy**( ) to move data within kernel space, since kernel space is never swapped out.

CONTEXT

bcopy( ) can be called from user or interrupt context.

EXAMPLE

An I / O request is made for data stored in a RAM disk. If the I / O operation is a read request, the data is copied from the RAM disk to a buffer (line 8). If it is a write request, the data is copied from a buffer to the RAM disk (line 15). **bcopy**( ) is used since both the RAM disk and the buffer are part of the kernel address space.

```
 1 #define RAMDNBLK          1000            /* blocks in the RAM disk */
 2 #define RAMDBSIZ          512             /* bytes per block */
 3 char ramdblks[RAMDNBLK][RAMDBSIZ];        /* blocks forming RAM */
                                             /* disk */
     ...
 4
 5 if  (bp->b_flags & B_READ)                /* if read request, copy data */
 6                                           /* from RAM disk data block */
 7                                           /* to system buffer */
 8         bcopy(&ramdblks[bp->b_blkno][0], bp->b_un.b_addr,
 9              bp->b_bcount);
10
11  else                                     /* else write request, */
12                                           /* copy data from a */
13                                           /* system buffer to RAM disk */
14                                           /* data block */
```

```
        bcopy(bp->b_un.b_addr, &ramdblks[bp->b_blkno][0],
                bp->b_bcount);
```

WARNINGS

The *from* and *to* addresses must be within the kernel space. No range checking is done. If an address outside of the kernel space is selected, the driver may corrupt the system in an unpredictable way.

SEE ALSO

copyin(9F), **copyout**(9F)

Writing Device Drivers

NAME

biodone – release buffer after buffer I/O transfer and notify blocked threads

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

void biodone(struct buf **bp***);**

ARGUMENTS

bp Pointer to a **buf**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

biodone( ) notifies blocked processes waiting for the I/O to complete, sets the **B_DONE** flag in the **b_flags** field of the **buf**(9S) structure, and releases the buffer if the I/O is asynchronous. **biodone**( ) is called by either the driver interrupt or **strategy**(9E) routines when a buffer I/O request is complete.

biodone( ) provides the capability to call a completion routine if *bp* describes a kernel buffer (the flag **B_KERNBUF** is set in the **b_flags** member). The address of the routine is specified in the **b_iodone** field of the **buf**(9S) structure. If such a routine is specified, **biodone**( ) calls it and returns without performing any other actions. Otherwise, it performs the steps above.

CONTEXT

biodone( ) can be called from user or interrupt context.

EXAMPLE

Generally, the first validation test performed by any block device **strategy**(9E) routine is a check for an end-of-file (EOF) condition. The **strategy**(9E) routine is responsible for determining an EOF condition when the device is accessed directly. If a **read**(2) request is made for one block beyond the limits of the device (line 10), it will report an EOF condition. Otherwise, if the request is outside the limits of the device, the routine will report an error condition. In either case, report the I/O operation as complete (line 27).

```
 1  #define RAMDNBLK    1000          /* Number of blocks in RAM disk */
 2  #define RAMDBSIZ     512          /* Number of bytes per block */
 3  char ramdblks[RAMDNBLK][RAMDBSIZ];  /* Array containing RAM disk */
 4
 5  static int
 6  ramdstrategy(struct buf *bp)
 7  {
 8          daddr_t blkno = bp->b_blkno;   /* get block number */
 9
10          if ((blkno < 0) || (blkno >= RAMDNBLK)) {
11                  /*
12                   * If requested block is outside RAM disk
13                   * limits, test for EOF which could result
14                   * from a direct (physio) request.
15                   */
16                  if ((blkno == RAMDNBLK) && (bp->b_flags & B_READ)) {
```

```
            /*
             * If read is for block beyond RAM disk
             * limits, mark EOF condition.
             */
             bp->b_resid = bp->b_bcount;/* compute return value */

        } else {                /* I/O attempt is beyond */
             bp->b_error = ENXIO;    /*  limits of RAM disk */
             bp->b_flags |= B_ERROR;   /* return error */
        }
        biodone(bp);          /* mark I/O complete (B_DONE) */
            /*
             * Wake any processes awaiting this I/O
             * or release buffer for asynchronous
             * (B_ASYNC) request.
             */
        return (0);
    }
...
```

SEE ALSO

read(2), **strategy**(9E), **biowait**(9F), **ddi_add_intr**(9F), **delay**(9F), **timeout**(9F), **untimeout**(9F), **buf**(9S)

Writing Device Drivers

NOTES

Drivers that use the **b_iodone** field of the **buf**(9S) structure to specify a substitute completion routine should save the value of **b_iodone** before changing it, and then restore the old value before calling **biodone**( ) to release the buffer.

NAME

bioerror – indicate error in buffer header

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>
#include <sys/ddi.h>

void bioerror(struct buf **bp***, int** *error***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

bp Pointer to the **buf**(9S) structure describing the transfer.

error Error number to be set, or zero to clear an error indication.

DESCRIPTION

If *error* is non-zero, **bioerror( )** indicates an error has occured in the **buf**(9S) structure. A subsequent call to **geterror**(9F) will return *error*.

If *error* is **0**, the error indication is cleared and a subsequent call to **geterror**(9F) will return **0**.

CONTEXT

bioerror( ) can be called from any context.

SEE ALSO

strategy(9E), **geterror**(9F), **getrbuf**(9F), **buf**(9S)

NAME

bioreset – reuse a private buffer header after I / O is complete

SYNOPSIS

#include <sys/buf.h>
#include <sys/ddi.h>

void bioreset(struct buf *∗bp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

bp Pointer to the **buf** (9S) structure.

DESCRIPTION

bioreset( ) is used by drivers that allocate private buffers with **getrbuf**(9F) and want to reuse them in multiple transfers before freeing them with **freerbuf**(9F). **bioreset( )** resets the buffer header to the state it had when initially allocated.

CONTEXT

bioreset( ) can be called from any context.

SEE ALSO

strategy(9E), **freerbuf**(9F), **getrbuf**(9F), **buf**(9S)

NOTES

bp must not describe a transfer in progress.

NAME

biowait – suspend processes pending completion of block I/O

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

int biowait(struct buf **bp***);**

ARGUMENTS

bp Pointer to the **buf** structure describing the transfer.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

Drivers allocating their own **buf** structures with **getrbuf**(9F) can use the **biowait**( ) function to suspend the current thread and wait for completion of the transfer.

Drivers must call **biodone**(9F) when the transfer is complete to notify the thread blocked by **biowait**( ). **biodone**( ) is usually called in the interrupt routine.

RETURN VALUES

0 on success

non-0 on I/O failure. **biowait**( ) calls **geterror**(9F) to retrieve the error number which it returns.

CONTEXT

biowait( ) can be called from user context only.

SEE ALSO

biodone(9F), **geterror**(9F), **getrbuf**(9F), **buf**(9S)

Writing Device Drivers

NAME

bp_mapin – allocate virtual address space

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

void bp_mapin(struct buf **bp***);**

ARGUMENTS

bp Pointer to the buffer header structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

bp_mapin( ) is used to map virtual address space to a page list maintained by the buffer header during a paged-I / O request. **bp_mapin**( ) allocates system virtual address space, maps that space to the page list, and returns the starting address of the space in the **bp->b_un.b_addr** field of the **buf**(9S) structure. Virtual address space is then deallocated using the **bp_mapout**(9F) function.

If a null page list is encountered, **bp_mapin**( ) returns without allocating space and no mapping is performed.

CONTEXT

bp_mapin( ) can be called from user context only.

SEE ALSO

bp_mapout(9F), **buf**(9S)

Writing Device Drivers

NAME

bp_mapout – deallocate virtual address space

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

void bp_mapout(struct buf **bp***);**

ARGUMENTS

bp Pointer to the buffer header structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

bp_mapout( ) deallocates system virtual address space allocated by a previous call to **bp_mapin**(9F).

CONTEXT

bp_mapout( ) can be called from user context only.

SEE ALSO

bp_mapin(9F), **buf**(9S)

Writing Device Drivers

NAME

brelse – return buffer to the free list

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

void brelse(struct buf *∗bp***);**

ARGUMENTS

bp Pointer to a **buf**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

brelse( ) returns a previously allocated buffer to the free buffer list. If any processes are waiting for this buffer to be released, or for any buffer to become available, they are notified.

CONTEXT

brelse( ) can be called from user or interrupt context.

SEE ALSO

strategy(9E), **biodone**(9F), **biowait**(9F), **clrbuf**(9F), **getrbuf**(9F)

Writing Device Drivers

WARNINGS

Do not call **brelse**( ) on buffers allocted by **getrbuf**(9F), or on buffers passed to the **strategy**(9E) routine.

BUGS

There is no sensible way for device drivers to use **brelse**( ).

NAME

btop – convert size in bytes to size in pages (round down)

SYNOPSIS

#include <sys/ddi.h>

unsigned long btop(unsigned long *numbytes***);**

ARGUMENTS

numbytes Number of bytes.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

btop( ) returns the number of memory pages that are contained in the specified number of bytes, with downward rounding in the case that the byte count is not a page multiple. For example, if the page size is 2048, then **btop(4096)** returns **2**, and **btop(4097)** returns **2** as well. **btop(0)** returns **0**.

RETURN VALUES

The return value is always the number of pages. There are no invalid input values, and therefore no error return values.

CONTEXT

btop( ) can be called from user or interrupt context.

SEE ALSO

btopr(9F), **ddi_btop**(9F), **ptob**(9F)

Writing Device Drivers

NAME

btopr – convert size in bytes to size in pages (round up)

SYNOPSIS

#include <sys/ddi.h>

unsigned long btopr(unsigned long *numbytes***);**

ARGUMENTS

numbytes Number of bytes.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

btopr( ) returns the number of memory pages contained in the specified number of bytes memory, rounded up to the next whole page. For example, if the page size is 2048, then **btopr(4096)** returns **2**, and **btopr(4097)** returns **3**.

RETURN VALUES

The return value is always the number of pages. There are no invalid input values, and therefore no error return values.

CONTEXT

btopr( ) can be called from user or interrupt context.

SEE ALSO

btop(9F), **ddi_btopr**(9F), **ptob**(9F)

Writing Device Drivers

NAME

bufcall – call a function when a buffer becomes available

SYNOPSIS

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

int bufcall(uint *size,* **int** *pri,* **void (***∗func)(long)***, long** *arg***);**

ARGUMENTS

size Number of bytes required for the buffer.

pri Priority of the **allocb**(9F) allocation request (not used).

func Function or driver routine to be called when a buffer becomes available.

arg Argument to the function to be called when a buffer becomes available.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

bufcall serves as a **timeout**(9F) call of indeterminate length. When a buffer allocation request fails, **bufcall**( ) can be used to schedule the routine *func*, to be called with the argument *arg* when a buffer becomes available. *func* may call **allocb** or it may do something else.

RETURN VALUES

If successful, **bufcall**( ) returns a **bufcall** id that can be used in a call to **unbufcall**( ) to cancel the request. If the **bufcall**( ) scheduling fails, *func* is never called and **0** is returned.

CONTEXT

bufcall( ) can be called from user or interrupt context.

EXAMPLE

The purpose of this **srv**(9E) service routine is to add a header to all **M_DATA** messages. Service routines must process all messages on their queues before returning, or arrange to be rescheduled.

While there are messages to be processed (line 13), check to see if it is a high priority message or a normal priority message that can be sent on (line 14). Normal priority message that cannot be sent are put back on the message queue (line 34). If the message was a high priority one, or if it was normal priority and **canputnext**(9F) succeeded, then send all but **M_DATA** messages to the next module with **putnext**(9F) (line 16).

For **M_DATA** messages, try to allocate a buffer large enough to hold the header (line 18). If no such buffer is available, the service routine must be rescheduled for a time when a buffer is available. The original message is put back on the queue (line 20) and **bufcall** (line 21) is used to attempt the rescheduling. It will succeed if the rescheduling succeeds, indicating that qenable will be called subsequently with the argument *q* once a buffer of the specified size (**sizeof (struct hdr)**) becomes available. If it does, **qenable**(9F) will put *q* on the list of queues to have their service routines called. If **bufcall** fails, **timeout**(9F) (line 22) is used to try again in about a half second.

If the buffer allocation was successful, initialize the header (lines 25–28), make the message type **M_PROTO** (line 29), link the **M_DATA** message to it (line 30), and pass it on (line 31).

Note that this example ignores the bookkeeping needed to handle **bufcall**( ) and **timeout**(9F) cancellation for ones that are still outstanding at close time.

```
struct hdr {
    unsigned int h_size;
    int        h_version;
};

void xxxsrv(q)
    queue_t *q;
{
    mblk_t *bp;
    mblk_t *mp;
    struct hdr *hp;

    while ((mp = getq(q)) != NULL) {        /* get next message */
        if (mp->b_datap->db_type >= QPCTL ||    /* if high priority */
            canputnext(q)) {        /* normal & can be passed */
            if (mp->b_datap->db_type != M_DATA)
                putnext(q, mp);        /* send all but M_DATA */
            else {
                bp = allocb(sizeof(struct hdr), BPRI_LO);
                if (bp == NULL) {        /* if unsuccessful */
                    putbq(q, mp);        /* put it back */
                    if (!bufcall(sizeof(struct hdr), BPRI_LO,
                      qenable, (long)q))    /* try to reschedule */
                        timeout(qenable, (caddr_t)q, drv_usectohz(500000));
                    return (0);
                }
                hp = (struct hdr *)bp->b_wptr;
                hp->h_size = msgdsize(mp);  /* initialize header */
                hp->h_version = 1;
                bp->b_wptr += sizeof(struct hdr);
                bp->b_datap->db_type = M_PROTO; /* make M_PROTO  */
                bp->b_cont = mp;        /* link it */
                putnext(q, bp);        /* pass it on */
            }
        } else {        /* normal priority, canputnext failed */
            putbq(q, mp);      /* put back on the message queue */
            return (0);
        }
    }
    return (0);
}
```

WARNINGS

Even when *func* is called by **bufcall**( ), **allocb**(9F) can fail if another module or driver had allocated the memory before *func* was able to call **allocb**(9F).

SEE ALSO

allocb(9F), **esballoc**(9F), **esbbcall**(9F), **testb**(9F), **timeout**(9F), **unbufcall**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

bzero – clear memory for a given number of bytes

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

void bzero(caddr_t *addr***, size_t** *bytes***);**

ARGUMENTS

addr Starting virtual address of memory to be cleared.

bytes The number of bytes to clear starting at *addr*.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

bzero( ) clears a contiguous portion of memory by filling it with zeros.

CONTEXT

bzero( ) can be called from user or interrupt context.

SEE ALSO

bcopy(9F), **clrbuf**(9F), **kmem_zalloc**(9F)

Writing Device Drivers

WARNINGS

The address range specified must be within the kernel space. No range checking is done. If an address outside of the kernel space is selected, the driver may corrupt the system in an unpredictable way.

NAME

canput – test for room in a message queue

SYNOPSIS

#include <sys/stream.h>

int canput(queue_t **q***);**

ARGUMENTS

q Pointer to the message queue.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

canput( ) searches through the stream (starting at *q*) until it finds a queue containing a service routine where the message can be enqueued, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if there is room for a message in the queue.

canputnext(*q***)** and **bcanputnext(***q***,** *pri***)** should always be used in preference to **canput(***q*→**q_next)** and **bcanput(***q*→**q_next,** *pri***)** respectively.

RETURN VALUES

1 If the message queue is not full.

0 If the queue is full.

CONTEXT

canput( ) can be called from user or interrupt context.

WARNINGS

Drivers are responsible for both testing a queue with **canput**( ) and refraining from placing a message on the queue if **canput**( ) fails.

SEE ALSO

bcanput(9F), **bcanputnext**(9F), **canputnext**(9F), **putbq**(9F), **putnext**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

canputnext, bcanputnext – test for room in next module's message queue

SYNOPSIS

#include <sys/stream.h>

int canputnext(queue_t **q***);**

int bcanputnext(queue_t **q***, unsigned char** *pri***);**

ARGUMENTS

q Pointer to a message queue belonging to the invoking module.

pri Minimum priority level.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

The invocation **canputnext(***q***);** is an atomic equivalent of the **canput(***q*→**q_next);** routine. That is, the STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing *q* through its **q_next** field and then invoking **canput**(9F) proceeds without interference from other threads.

bcanputnext(*q***,** *pri***);** is the equivalent of the **bcanput(***q*→**q_next,** *pri***);** routine.

canputnext(*q***);** and **bcanputnext(***q***,** *pri***);** should always be used in preference to **canput(***q*→**q_next);** and **bcanput(***q*→**q_next,** *pri***);** respectively.

See **canput**(9F) and **bcanput**(9F) for further details.

RETURN VALUES

1 If the message queue is not full.

0 If the queue is full.

CONTEXT

canputnext( ) and **bcanputnext**( ) can be called from user or interrupt context.

WARNINGS

Drivers are responsible for both testing a queue with **canputnext**( ) or **bcanputnext**( ) and refraining from placing a message on the queue if the queue is full.

SEE ALSO

bcanput(9F), **canput**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

clrbuf – erase the contents of a buffer

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

void clrbuf(struct buf *∗bp***);**

ARGUMENTS

bp Pointer to the **buf**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

clrbuf( ) zeros a buffer and sets the **b_resid** member of the **buf**(9S) structure to **0**. Zeros are placed in the buffer starting at **bp->b_un.b_addr** for a length of **bp->b_bcount** bytes. **b_un.b_addr** and **b_bcount** are members of the **buf**(9S) data structure.

CONTEXT

clrbuf( ) can be called from user or interrupt context.

SEE ALSO

brelse(9F), **buf**(9S)

Writing Device Drivers

NAME

cmn_err, vcmn_err – display an error message or panic the system

SYNOPSIS

#include <sys/cmn_err.h>

void cmn_err(int *level***, char** **format***, ...);**

void vcmn_err(int *level***, char** **format***, va_list** *ap***);**

ARGUMENTS

cmn_err( )

level A constant indicating the severity of the error condition. The four severity levels are:

CE_CONT Used to continue another message or to display an informative message not connected with an error.

CE_NOTE Used to display a message preceded with **NOTICE**. This message is used to report system events that do not necessarily require user action, but may interest the system administrator. For example, a message saying that a sector on a disk needs to be accessed repeatedly before it can be accessed correctly might be noteworthy.

CE_WARN Used to display a message preceded with **WARNING**. This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.

CE_PANIC Used to display a message preceded with **PANIC** or **DOUBLE PANIC**, and to panic the system. Drivers should specify this level only under the most severe conditions or when debugging a driver. A valid use of this level is when the system cannot continue to function. If the error is recoverable, or not essential to continued system operation, do not panic the system. This level halts multiuser processing.

format The message to be displayed. By default, the message is sent both to the system console and to the system buffer. If the first character in *format* is an '!' (exclamation point), the message goes only to the system buffer. If the first character in *format* is a '^' (circumflex), the message goes only to the console. If the first character is a '?' (question mark), and level is **CE_CONT**, the message is always sent to the system buffer, but is only written to the console when the system has been booted in verbose mode. See **kernel**(1M). If neither condition is met, the '?' character has no effect and is simply ignored. Except for the first character, the rules for *format* are the same as those for **printf**(3S) strings. To display the contents of the system buffer, use the **dmesg**(1M) command.

cmn_err appends a **\n** to each *format*, except when *level* is **CE_CONT**.

Valid conversion specifications are **%s**, **%u**, **%d**, **%b**, **%o**, and **%x**.

The **%b** conversion specification allows bit values to be printed meaningfully.

Each **%b** takes an integer value and a format string from the argument list. The first character of the format string should be the output base encoded as a control character. This base is used to print the integer argument. The remaining groups of characters in the format string consist of a bit number (between 1 and 32, also encoded as a control character) and the next characters (up to the next control character or '\0') give the name of the bit field. The string corresponding to the bit fields set in the integer argument is printed after the numerical value. See the examples below.

cmn_err( ) is otherwise similar to the **printf**(3S) library subroutine in displaying messages.

vcmn_err( )

vcmn_err( ) takes *level* and *format* as described for **cmn_err**( ), but its third argument is different:

ap The var arg list passed to the function.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

cmn_err( )

cmn_err( ) displays a specified message on the console. **cmn_err**( ) can also panic the system.

At times, a driver may encounter error conditions requiring the attention of a primary or secondary system console monitor. These conditions may mean halting multiuser processing; however, this must be done with caution. Except during the debugging stage, a driver should never stop the system.

cmn_err( ) with the **CE_CONT** argument can be used by driver developers as a driver code debugging tool. However, using **cmn_err**( ) in this capacity can change system timing characteristics.

If **CE_PANIC** is set, **cmn_err**( ) stops the machine.

vcmn_err( )

vcmn_err( ) is identical to **cmn_err**( ) except that its last argument *ap* is a pointer to a list of arguments.

RETURN VALUES

None. However, if an unknown *level* is passed to **cmn_err**( ), the following panic error message is displayed:

PANIC: unknown level in cmn_err (level= *level* **, msg=** *format***)**

CONTEXT

cmn_err( ) can be called from user or interrupt context.

EXAMPLES

This first example shows how **cmn_err**( ) can record tracing and debugging information only in the system buffer (lines 15 and 16); display problems with a device only on the system console (line 21); or stop the system if a required device malfunctions (line 27).

```
1  struct  device {
2        int     control;
3        int     status;
```

```
        int      error;
        short    recv_char;
        short    xmit_char;
};

extern struct device xx_addr[];
extern int          xx_cnt;
 . . .
register struct device *rp;
rp = xx_addr[(getminor(dev) >> 4) & 0xf];    /* get dev registers */

#ifdef DEBUG            /* in debugging mode, log function call */
    cmn_err(CE_NOTE, "!xx_open function call, dev = 0x%x", dev);
    cmn_err(CE_CONT, "! flag = 0x%x", flag);     /* continue msg */
#endif  /* end DEBUG */

/* display device power failure on system console */
  if ((rp->status & POWER) == OFF)
    cmn_err(CE_WARN, "xx_open: Power is OFF on device %d port %d",
           ((getminor(dev) >> 4) & 0xf), (getminor(dev) & 0xf));

/* halt system if root device has bad VTOC */
  if (rp->error == BADVTOC  &&  dev == rootdev)
    cmn_err(CE_PANIC, "xx_open: Bad VTOC on root device");
```

The second example shows how to use the **%b** conversion specification. Because of the leading '?' character in the format string, this message will always be logged, but it will only be displayed when the kernel is booted in verbose mode.

```
cmn_err(CE_CONT, "?reg=0x%b\n", regval, "\020\3Intr\2Err\1Enable");
```

When *regval* is set to (decimal) **13**, the following message would be printed:

```
reg=0xd<Intr,,Enable>
```

SEE ALSO

dmesg(1M), **kernel**(1M), **printf**(3S), **print**(9E), **ddi_report_dev**(9F)

Writing Device Drivers

NOTES

cmn_err( ) does not accept length specifications in conversion specifications. For example, **%3d** is ignored.

BUGS

See chapter 12, "Debugging" in *Writing Device Drivers*.

NAME

condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS

#include <sys/ksynch.h>

void cv_init(kcondvar_t **cvp*, **char** **name*, **kcv_type_t** *type*, **void** **arg***);**

void cv_destroy(kcondvar_t **cvp***);**

void cv_wait(kcondvar_t **cvp*, **kmutex_t** **mp***);**

void cv_signal(kcondvar_t **cvp***);**

void cv_broadcast(kcondvar_t **cvp***);**

int cv_wait_sig(kcondvar_t **cvp*, **kmutex_t** **mp***);**

int cv_timedwait(kcondvar_t **cvp*, **kmutex_t** **mp*, **long** *timeout***);**

int cv_timedwait_sig(kcondvar_t **cvp*, **kmutex_t** **mp*, **long** *timeout***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

cvp A pointer to an abstract data type **kcondvar_t**.

mp A pointer to a mutual exclusion lock (**kmutex_t**), initialized by **mutex_init**(9F) and held by the caller.

name A name for the condition variable, used in statistics and debugging.

type The constant **CV_DRIVER**.

arg A type-specific argument, drivers should pass arg as **NULL**.

timeout A time, in absolute ticks since boot, when **cv_timedwait**( ) or **cv_timedwait_sig**( ) should return.

DESCRIPTION

Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling **cv_init**( ), and must be deallocated by calling **cv_destroy**( ).

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, **cv_wait**( ) is called with a related condition variable and the mutex. At some later point in time, another thread would aquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with **cv_signal**( ) or **cv_broadcast**( ), and then release the mutex.

cv_wait( ) suspends the calling thread and exits the mutex atomically so that another thread which holds the mutex cannot signal on the condition variable until the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal( ) signals the condition and wakes one blocked thread. All blocked threads can be unblocked by calling **cv_broadcast**( ). You must aquire the mutex passed into **cv_wait**( ) before calling **cv_signal**( ) or **cv_broadcast**( ).

The function **cv_wait_sig**( ) is similar to **cv_wait**( ) but returns **0** if a signal (for example, by **kill**(2)) is sent to the thread. In any case, the mutex is reacquired before returning.

The function **cv_timedwait**( ) is similar to **cv_wait**( ), except that it returns –**1** without the condition being signaled after the timeout time has been reached.

The function **cv_timedwait_sig**( ) is similar to **cv_timedwait**( ), and **cv_wait_sig**( ), except that it returns –**1** without the condition being signaled after the timeout time has been reached, or **0** if a signal (for example, by **kill**(2)) is sent to the thread.

For both **cv_timedwait**( ) and **cv_timedwait_sig**( ), time is in absolute clock ticks since the last system reboot. The current time may be found by calling **drv_getparm**(9F) with the argument **LBOLT**.

RETURN VALUES

0 For **cv_wait_sig**( ) and **cv_timedwait_sig**( ) indicates that the condition was not necessarily signaled and the function returned because a signal (as in **kill**(2)) was pending.

-1 For **cv_timedwait**( ) and **cv_timedwait_sig**( ) indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.

> **0** For **cv_wait_sig**( ), **cv_timedwait**( ) or **cv_timedwait_sig**( ) indicates that the condition was met and the function returned due to a call to **cv_signal**( ) or **cv_broadcast**( ).

CONTEXT

These functions can be called from user, kernel or interrupt context. In most cases, however, **cv_wait**( ), **cv_timedwait**( ), **cv_wait_sig**( ), and **cv_timedwait_sig**( ) should be called from user context only.

EXAMPLES

Here the condition being waited for is a flag value in a driver's unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
        cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```
mutex_enter(&un->un_lock);
un->un_flag &= ˜UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

SEE ALSO

kill(2), **drv_getparm**(9F), **mutex**(9F), **mutex_init**(9F)

Writing Device Drivers

NAME

copyb – copy a message block

SYNOPSIS

#include <sys/stream.h>

mblk_t *copyb(mblk_t ***bp*****);**

ARGUMENTS

bp   Pointer to the message block from which data is copied.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

copyb( ) allocates a new message block, and copies into it the data from the block that *bp* denotes. The new block will be at least as large as the block being copied. **copyb**( ) uses the **b_rptr** and **b_wptr** members of *bp* to determine how many bytes to copy.

RETURN VALUES

If successful, **copyb**( ) returns a pointer to the newly allocated message block containing the copied data. Otherwise, it returns a **NULL** pointer.

CONTEXT

copyb( ) can be called from user or interrupt context.

EXAMPLES

For each message in the list, test to see if the downstream queue is full with the **canputnext**(9F) function (line 21). If it is not full, use **copyb**(9F) to copy a header message block, and **dupmsg**(9F) to duplicate the data to be retransmitted. If either operation fails, reschedule a timeout at the next valid interval.

Update the new header block with the correct destination address (line 34), link the message to it (line 35), and send it downstream (line 36). At the end of the list, reschedule this routine.

```
 1  struct retrans {
 2          mblk_t              *r_mp;
 3          long                r_address;
 4          queue_t             *r_outq;
 5          struct retrans      *r_next;
 6  };
 7
 8  struct protoheader {
        . . .
 9      long                    h_address;
        . . .
10  };
11
12  mblk_t *header;
13
14  void
15  retransmit(struct retrans *ret)
16  {
17          mblk_t *bp, *mp;
```

```
        struct protoheader *php;

        while (ret) {
            if (!canputnext(ret->r_outq)) {         /* no room */
                ret = ret->r_next;
                continue;
            }
            bp = copyb(header);                     /* copy header msg. block */
            if (bp == NULL)
                break;
            mp = dupmsg(ret->r_mp);                 /* duplicate data */
            if (mp == NULL) {                       /* if unsuccessful */
                freeb(bp);                          /* free the block */
                break;
            }
            php = (struct protoheader *)bp->b_rptr;
            php->h_address = ret->r_address;        /* new header */
            bp->bp_cont = mp;                       /* link the message */
            putnext(ret->r_outq, bp);               /* send downstream */
            ret = ret->r_next;
        }
        /* reschedule */
        (void) timeout(retransmit, (caddr_t)ret, RETRANS_TIME);
}
```

SEE ALSO

allocb(9F), **canputnext**(9F), **dupmsg**(9F)

Writing Device Drivers
STREAMS Programmer’s Guide

NAME

copyin – copy data from a user program to a driver buffer

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

int copyin(caddr_t *userbuf***, caddr_t** *driverbuf***, size_t** *cn***);**

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

ARGUMENTS

userbuf User program source address from which data is transferred.

driverbuf Driver destination address to which data is transferred.

cn Number of bytes transferred.

DESCRIPTION

copyin( ) copies data from a user program source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address.

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move according to address alignment.

RETURN VALUES

Under normal conditions a **0** is returned indicating a successful copy. Otherwise, a **-1** is returned if one of the following occurs:

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

If a **-1** is returned to the caller, driver entry point routines should return **EFAULT**.

CONTEXT

copyin( ) can be called from user context only.

EXAMPLES

A driver **ioctl**(9E) routine (line 9) can be used to get or set device attributes or registers. In the **XX_GETREGS** condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned.

```
1 struct  device {     /* layout of physical device registers  */
2       int     control;    /* physical device control word  */
3       int     status;     /* physical device status word  */
4       short    recv_char;  /* receive character from device */
5       short    xmit_char;   /* transmit character to device  */
6 }; /* end device */
7
8 extern struct device xx_addr[]; /* phys. device regs. location */
   . . .
```

```
xx_ioctl(dev, cmd, arg, mode, cred_p, rval_p)
        dev_t   dev;
        int     cmd, arg;
            ...
{
    register struct device *rp = &xx_addr[getminor(dev) >> 4];
    switch (cmd) {

     case XX_SETREGS:     /* copy device regs. to user program */
         if (copyin((caddr_t)arg, (caddr_t)rp, sizeof(struct device)))
            return(EFAULT);
         break;
```

SEE ALSO

bcopy(9F), **copyout**(9F), **ddi_copyin**(9F), **ddi_copyout**(9F), **uiomove**(9F).

Writing Device Drivers

NOTES

Driver writers who intend to support layered ioctls in their **ioctl**(9E) routines should use **ddi_copyin**(9F) instead.

Driver defined locks should not be held across calls to this function.

NAME

copymsg – copy a message

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗copymsg(mblk_t ∗*mp*);

ARGUMENTS

mp Pointer to the message to be copied.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

copymsg( ) forms a new message by allocating new message blocks, and copying the contents of the message referred to by *mp* (using the **copyb**(9F) function). It returns a pointer to the new message.

RETURN VALUES

If the copy is successful, **copymsg**( ) returns a pointer to the new message. Otherwise, it returns a **NULL** pointer.

CONTEXT

copymsg( ) can be called from user or interrupt context.

EXAMPLES

The routine **lctouc**( ) converts all the lowercase ASCII characters in the message to uppercase. If the reference count is greater than one (line 8), then the message is shared, and must be copied before changing the contents of the data buffer. If the call to the **copymsg**(9F) function fails (line 9), return **NULL** (line 10), otherwise, free the original message (line 11). If the reference count was equal to **1**, the message can be modified. For each character (line 16) in each message block (line 15), if it is a lowercase letter, convert it to an uppercase letter line 18). A pointer to the converted message is returned (line 21).

```
 1 mblk_t *lctouc(mp)
 2         mblk_t *mp;
 3 {
 4         mblk_t *cmp;
 5         mblk_t *tmp;
 6         unsigned char *cp;
 7
 8         if (mp->b_datap->db_ref > 1) {
 9                 if ((cmp = copymsg(mp)) == NULL)
10                         return (NULL);
11                 freemsg(mp);
12         } else {
13                 cmp = mp;
14         }
15         for (tmp = cmp; tmp; tmp = tmp->b_next) {
16                 for (cp = tmp->b_rptr; cp < tmp->b_wptr; cp++) {
17                         if ((*cp <= 'z') && (*cp >= 'a'))
18                                 *cp -= 0x20;
```

```
                }
        }
        return(cmp);
}
```

SEE ALSO

allocb(9F), **copyb**(9F), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

copyout – copy data from a driver to a user program

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

int copyout(caddr_t *driverbuf*, **caddr_t** *userbuf*, **size_t** *cn***);**

ARGUMENTS

driverbuf Source address in the driver from which the data is transferred.

userbuf Destination address in the user program to which the data is transferred.

cn Number of bytes moved.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

copyout( ) copies data from driver buffers to user data space.

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

RETURN VALUES

Under normal conditions a **0** is returned to indicate a successful copy. Otherwise, a **-1** is returned if one of the following occurs:

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

If a **-1** is returned to the caller, driver entry point routines should return **EFAULT**.

CONTEXT

copyout( ) can be called from user context only.

EXAMPLES

A driver **ioctl**(9E) routine (line 9) can be used to get or set device attributes or registers. In the **XX_GETREGS** condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned.

```
 1  struct device  {                          /* layout of physical device registers  */
 2      int       control;                    /* physical device control word  */
 3      int       status;                     /* physical device status word   */
 4      short     recv_char;                  /* receive character from device */
 5      short     xmit_char;                  /* transmit character to device  */
 6  }; /* end device */
 7
 8  extern struct device xx_addr[];           /* phys. device regs. location */
     . . .
 9  xx_ioctl(dev, cmd, arg, mode, cred_p, rval_p)
10      dev_t     dev;
```

```
      int        cmd, arg;
          ...
{
      register struct device *rp = &xx_addr[getminor(dev) >> 4];
      switch (cmd)  {
          case XX_GETREGS:                /* copy device regs. to user program */
               if (copyout((caddr_t)rp, (caddr_t)arg, sizeof(struct device)))
                   return(EFAULT);
               break;
```

SEE ALSO

bcopy(9F), **copyin**(9F), **ddi_copyin**(9F), **ddi_copyout**(9F), **uiomove**(9F).

Writing Device Drivers

NOTES

Driver writers who intend to support layered ioctls in their **ioctl**(9E) routines should use **ddi_copyout**(9F) instead.

Driver defined locks should not be held across calls to this function.

NAME

datamsg – test whether a message is a data message

SYNOPSIS

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

int datamsg(unsigned char *type***);**

ARGUMENTS

type The type of message to be tested. The **db_type** field of the **datab**(9S) structure contains the message type. This field may be accessed through the message block using **mp->b_datap->db_type**.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

datamsg( ) tests the type of message to determine if it is a data message type (**M_DATA**, **M_DELAY**, **M_PROTO**, or **M_PCPROTO**).

RETURN VALUES

datamsg returns **1** if the message is a data message; and **0** otherwise.

CONTEXT

datamsg( ) can be called from user or interrupt context.

EXAMPLES

The **put**(9E) routine enqueues all data messages for handling by the **srv**(9E) (service) routine. All non-data messages are handled in the **put**(9E) routine.

```
xxxput(q, mp)
      queue_t *q;
      mblk_t *mp;
{
      if (datamsg(mp->b_datap->db_type)) {
              putq(q, mp);
              return;
      }
      switch (mp->b_datap->db_type) {
      case M_FLUSH:
              . . .
      }
}
```

SEE ALSO

put(9E), **srv**(9E), **allocb**(9F), **datab**(9S), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

ddi_add_intr, ddi_remove_intr – add and remove an interrupt handler

SYNOPSIS

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

int ddi_add_intr(dev_info_t **dip*, **u_int** *inumber*, **ddi_iblock_cookie_t** **iblock_cookiep*,
ddi_idevice_cookie_t **idevice_cookiep*, **u_int (****int_handler*)**(caddr_t),**
caddr_t *int_handler_arg***);**

void ddi_remove_intr(dev_info_t **dip*, **u_int** *inumber*, **ddi_iblock_cookie_t**
*iblock_cookie***);**

ARGUMENTS

ddi_add_intr( )

dip	Pointer to **dev_info** structure.
inumber	Interrupt number.
iblock_cookiep	Pointer to an interrupt block cookie.
idevice_cookiep	Pointer to an interrupt device cookie.
int_handler	Pointer to interrupt handler.
int_handler_arg	Argument for interrupt handler.

ddi_remove_intr( )

dip	Pointer to **dev_info** structure.
inumber	Interrupt number.
iblock_cookie	Block cookie which identifies the interrupt handler to be removed.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_add_intr( ) adds an interrupt handler to the system. The interrupt number *inumber* determines which interrupt the handler will be associated with. The parameter *inumber* is associated with information provided either by the device (see **sbus**(4)) or the hardware configuration file (see **vme**(4) and **driver.conf**(4)). If only one interrupt is associated with the device, *inumber* should be 0.

On a successful return, *iblock_cookiep* contains information needed for initializing mutexes associated with this interrupt specification (see **mutex_init**(9F)). If *iblock_cookiep* is set to NULL, no value will be returned.

On a successful return, *idevice_cookiep* contains a pointer to a structure containing information useful for some devices that have programmable interrupts. The **idev_priority** field of the returned structure contains the bus interrupt priority level and the **idev_vector** field contains the vector number for vectored bus architectures such as VMEbus. If *idevice_cookiep* is set to NULL, no value is returned.

The routine *intr_handler*, with its argument *int_handler_arg*, is called upon receipt of the appropriate interrupt. The interrupt handler should return **DDI_INTR_CLAIMED** if the interrupt was claimed, **DDI_INTR_UNCLAIMED** otherwise.

If successful, **ddi_add_intr**( ) will return **DDI_SUCCESS**; if the interrupt information cannot be found, it will return **DDI_INTR_NOTFOUND.**

ddi_remove_intr( ) removes an interrupt handler from the system. Unloadable drivers should call this routine during their **detach**(9E) routine to remove their interrupt handler from the system.

The device interrupt routine for this instance of the device will not execute after **ddi_remove_intr**( ) returns. **ddi_remove_intr**( ) may need to wait for the device interrupt routine to complete before returning. Therefore, locks acquired by the interrupt handler should not be held across the call to **ddi_remove_intr**( ) or deadlock may result.

RETURN VALUES

ddi_add_intr( ) returns:

DDI_SUCCESS	on success.
DDI_INTR_NOTFOUND	on failure to find the interrupt.

CONTEXT

ddi_add_intr( ) and **ddi_remove_intr**( ) can be called from user or interrupt context.

SEE ALSO

driver.conf(4), **sbus**(4), **vme**(4), **attach**(9E), **detach**(9E), **ddi_intr_hilevel**(9F), **mutex_init**(9F)

Writing Device Drivers

BUGS

The *idevice_cookiep* should really point to a data structure that is specific to the bus architecture that the device operates on. Currently only VMEbus and SBus are supported and a single data structure is used to describe both.

It is possible that a driver's interrupt handler will be called immediately *after* the driver has called **ddi_add_intr**( ) but *before* the driver has had an opportunity to initialize its mutexes. This can happen when an interrupt for a different device occurs on the same interrupt level. If the interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

The solution to this problem is to add a temporary interrupt handler using **ddi_add_intr**( ). The temporary interrupt routine must be a function that performs no action, such as **nulldev**(9F). This allows the driver to obtain the interrupt block cookie for the interrupt, which it can then use to initialize any mutexes. After the mutexes are initialized, the temporary interrupt handler can be removed, and the real one installed. **nulldev**(9F) can be used as the temporary interrupt handler, though it needs to be cast properly.

NAME

ddi_add_softintr, ddi_remove_softintr, ddi_trigger_softintr – add, remove or trigger a soft interrupt

SYNOPSIS

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

int ddi_add_softintr(dev_info_t **dip,* **int** *preference,* **ddi_softintr_t** **idp,*
ddi_iblock_cookie_t **ibcp,* **ddi_idevice_cookie_t** **idcp,*
u_int(**int_handler*)**(caddr_t** *int_handler_arg*), **caddr_t** *int_handler_arg***)**
void ddi_remove_softintr(ddi_softintr_t *id***)**
void ddi_trigger_softintr(ddi_softintr_t *id***)**

ARGUMENTS

ddi_add_softintr( )

dip	Pointer to **dev_info** structure.
preference	A hint value describing the type of soft interrupt to generate.
idp	Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.
ibcp	Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
idcp	Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.
int_handler	Pointer to interrupt handler.
int_handler_arg	Argument for interrupt handler.

ddi_remove_softintr( )

id	The identifier specifying which soft interrupt handler to remove.

ddi_trigger_softintr( )

id	The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_add_softintr( ) adds a soft interrupt to the system. The user specified hint *preference* identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The possible values for *preference* are:

DDI_SOFTINT_LOW	Low priority soft interrupt
DDI_SOFTINT_MED	Medium priority soft interrupt
DDI_SOFTINT_HIGH	High priority soft interrupt

The value returned in location pointed at by *idp* is the soft interrupt identifier. This value is used in later calls to **ddi_remove_softintr**( ) and **ddi_trigger_softintr**( ) to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by *ibcp* is an interrupt block cookie which contains information needed for initializing mutexes associated with this soft interrupt (see **mutex_init**(9F). If the interrupt cookie pointer is set to **NULL** no value will be returned.

The value returned in the location pointed at by *idcp* is an interrupt device cookie which contains the machine specific bits used by the system to program a soft interrupt. This value is currently not useful to device drivers and is available only for future extensions to the DDI/DKI. If the device cookie pointer is set to **NULL** no value will be returned.

The routine *intr_handler*, with its argument *int_handler_arg*, is called upon receipt of appropriate soft interrupt. The interrupt handler should return **DDI_INTR_CLAIMED** if the interrupt was claimed, **DDI_INTR_UNCLAIMED** otherwise.

If successful, **ddi_add_softintr**( ) will return **DDI_SUCCESS**; if the interrupt information cannot be found, it will return **DDI_FAILURE**.

ddi_remove_softintr( ) removes a soft interrupt from the system. The soft interrupt identifier *id*, which was returned from a call to **ddi_add_softintr**( ), is used to determine which soft interrupt and which soft interrupt handler to remove. Unloadable drivers should call this routine to detach themselves from the system.

ddi_trigger_softintr( ) triggers a soft interrupt. The soft interrupt identifier *id*, which was returned from a call to **ddi_add_softintr**( ), is used to determine which soft interrupt to trigger and subsequently which soft interrupt handler to call. This function is used by device drivers when they wish to trigger a soft interrupt which they had set up using **ddi_add_softintr**( ).

RETURN VALUES

ddi_add_softintr( ) returns:

DDI_SUCCESS	on success
DDI_FAILURE	on failure

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

ddi_add_intr(9F), **ddi_remove_intr**(9F), **mutex_init**(9F)

Writing Device Drivers

NAME

ddi_btop, ddi_btopr, ddi_ptob – page size conversions

SYNOPSIS

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

unsigned long ddi_btop(dev_info_t **dip*, **unsigned long** *bytes***);**

unsigned long ddi_btopr(dev_info_t **dip*, **unsigned long** *bytes***);**

unsigned long ddi_ptob(dev_info_t **dip*, **unsigned long** *pages***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

This set of routines use the parent nexus driver to perform conversions in page size units.

ddi_btop( ) converts the given number of bytes to the number of memory pages that it corresponds to, rounding down in the case that the byte count is not a page multiple.

ddi_btopr( ) converts the given number of bytes to the number of memory pages that it corresponds to, rounding up in the case that the byte count is not a page multiple.

ddi_ptob( ) converts the given number of pages to the number of bytes that it corresponds to.

Because bus nexus may possess their own hardware address translation facilities, these routines should be used in preference to the corresponding DDI / DKI routines **btop**(9F), **btopr**(9F), and **ptob**(9F), which only deal in terms of the pagesize of the main system MMU.

RETURN VALUES

ddi_btop( ) and **ddi_btopr**( ) return the number of corresponding pages. **ddi_ptob**( ) returns the corresponding number of bytes. There are no error return values.

CONTEXT

This function can be called from user or interrupt context.

EXAMPLE

This example finds the size (in bytes) of one page:

```
pagesize = ddi_ptob(dip, 1L);
```

SEE ALSO

btop(9F), **btopr**(9F), **ptob**(9F)

Writing Device Drivers

NAME

ddi_copyin – copy data to a driver buffer

SYNOPSIS

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

int ddi_copyin(caddr_t *buf***, caddr_t** *driverbuf***, size_t** *cn***, int** *flags***);**

ARGUMENTS

buf Source address from which data is transferred.

driverbuf Driver destination address to which data is transferred.

cn Number of bytes transferred.

flags Set of flag bits that provide address space information about *buf*.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

This routine is designed for use in driver **ioctl**(9E) routines for drivers that support layered ioctls. **ddi_copyin**( ) copies data from a source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address.

The *flags* argument is used to determine the address space information about *buf*. If the FKIOCTL flag is set, this indicates that *buf* is a kernel address, and **ddi_copyin**( ) behaves like **bcopy**(9F). Otherwise *buf* is interpreted as a user buffer address, and **ddi_copyin**( ) behaves like **copyin**(9F).

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move according to address alignment.

RETURN VALUES

ddi_copyin( ) returns **0**, indicating a successful copy. It returns –**1** if one of the following occurs:

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

If a –**1** is returned to the caller, driver entry point routines should return EFAULT.

EXAMPLES

A driver **ioctl**(9E) routine (line 11) can be used to get or set device attributes or registers. In the **XX_GETREGS** condition (line 24), the driver copies the current device register values to another data area (line 25). If the specified argument contains an invalid address, an error code is returned.

```
struct device {     /* layout of physical device registers */
    int     control;  /* physical device control word */
    int     status;   /* physical device status word */
    short   recv_char;/* receive character from device */
    short   xmit_char;/* transmit character to device */
};

struct device_state {
    volatile struct device *regsp; /* pointer to device registers */
   . . .
};

static void *statep;    /* for soft state routines */

xxioctl(dev_t dev, int cmd, int arg, int mode,
    cred_t *cred_p, int *rval_p)
{
    struct device_state *sp;
    volatile struct device *rp;
    int instance;

    instance = getminor(dev) >> 4;
    sp = ddi_get_soft_state(statep, instance);
    if (sp == NULL)
        return (ENXIO);
    rp = sp->regsp;
   . . .
    switch (cmd)  {

    case XX_SETREGS:    /* copy device regs. to caller */
          if (ddi_copyin((caddr_t)rp, (caddr_t)arg,
              sizeof (struct device), mode) != 0) {
                  return (EFAULT);
          }
```

CONTEXT

ddi_copyin( ) can be called from user context only.

SEE ALSO

ioctl(9E), **bcopy**(9F), **copyin**(9F), **copyout**(9F), **ddi_copyout**(9F), **uiomove**(9F)

Writing Device Drivers

NOTES

The value of the *flags* argument to **ddi_copyin**( ) should be passed through directly from the *mode* argument of **ioctl**( ) untranslated.

Driver defined locks should not be held across calls to this function.

NAME

ddi_copyout – copy data from a driver

SYNOPSIS

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

int ddi_copyout(caddr_t *driverbuf*, **caddr_t** *buf*, **size_t** *cn*, **int** *flags***);**

ARGUMENTS

driverbuf Source address in the driver from which the data is transferred.

buf Destination address to which the data is transferred.

cn Number of bytes to copy.

flags Set of flag bits that provide address space information about *buf*.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

This routine is designed for use in driver **ioctl**(9E) routines for drivers that support layered ioctls. **ddi_copyout**( ) copies data from driver buffers to a destination address, *buf*.

The *flags* argument is used to determine the address space information about *buf*. If the FKIOCTL flag is set, this indicates that *buf* is a kernel address, and **ddi_copyout**( ) behaves like **bcopy**(9F). Otherwise *buf* is interpreted as a user buffer address, and **ddi_copyout**( ) behaves like **copyout**(9F).

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

RETURN VALUES

Under normal conditions a **0** is returned to indicate a successful copy. Otherwise, a **-1** is returned if one of the following occurs:

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user address, such as a user area or stack area
- invalid address that would have resulted in data being copied into the user block

If a **-1** is returned to the caller, driver entry point routines should return **EFAULT**.

CONTEXT

ddi_copyout( ) can be called from user context only.

EXAMPLES

A driver **ioctl**(9E) routine (line 11) can be used to get or set device attributes or registers. In the **XX_GETREGS** condition (line 24), the driver copies the current device register values to another data area (line 25). If the specified argument contains an invalid address, an error code is returned.

```
1  struct device {     /* layout of physical device registers */
2     int     control;  /* physical device control word */
3     int     status;   /* physical device status word */
```

```
    short   recv_char;/* receive character from device */
    short   xmit_char;/* transmit character to device  */
};

struct device_state {
    volatile struct device *regsp; /* pointer to device registers */
    . . .
};

static void *statep;     /* for soft state routines */

xxioctl(dev_t dev, int cmd, int arg, int mode,
    cred_t *cred_p, int *rval_p)
{
    struct device_state *sp;
    volatile struct device *rp;
    int instance;

    instance = getminor(dev) >> 4;
    sp = ddi_get_soft_state(statep, instance);
    if (sp == NULL)
        return (ENXIO);
    rp = sp->regsp;

    . . .
    switch (cmd)  {

    case XX_GETREGS:     /* copy device regs. to caller */
          if (ddi_copyout((caddr_t)rp, (caddr_t)arg,
              sizeof (struct device), mode) != 0) {
                  return (EFAULT);
          }
```

SEE ALSO

bcopy(9F), **copyin**(9F), **copyout**(9F), **ddi_copyin**(9F), **uiomove**(9F)

Writing Device Drivers

NOTES

The value of the *flags* argument to **ddi_copyout**( ) should be passed through directly from the *mode* argument of **ioctl**( ) untranslated.

Driver defined locks should not be held across calls to this function.

NAME

ddi_create_minor_node – create a minor node for this device

SYNOPSIS

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

int ddi_create_minor_node(dev_info_t **dip*, **char** **name*,
    int *spec_type*, **int** *minor_num*, **char** **node_type*,
    int *is_clone***);**

ARGUMENTS

dip — A pointer to the device's **dev_info** structure.

name — The name of this particular minor device.

spec_type — **S_IFCHR** or **S_IFBLK** for character or block minor devices respectively.

minor_num — The minor number for this particular minor device.

node_type — Any string that uniquely identifies the type of node. The following predefined node types are provided with this release:

DDI_NT_SERIAL	For serial ports
DDI_NT_SERIAL_MB	For on board serial ports
DDI_NT_SERIAL_DO	For dial out ports
DDI_NT_SERIAL_MB_DO	For on board dial out ports
DDI_NT_BLOCK	For hard disks
DDI_NT_BLOCK_CHAN	For hard disks with channel or target numbers
DDI_NT_CD	For CDROM drives
DDI_NT_CD_CHAN	For CDROM drives with channel or target numbers
DDI_NT_FD	For floppy disks
DDI_NT_TAPE	For tape drives
DDI_NT_NET	For network devices
DDI_NT_DISPLAY	For display devices
DDI_PSEUDO	For pseudo devices

is_clone — If the device is a clone device then this flag is set to **CLONE_DEV** else it is set to 0.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_create_minor_node( ) provides the necessary information to enable the system to create the **/dev** and **/devices** hierarchies. The *name* is used to create the minor name of the block or character special file under the **/devices** hierarchy. At sign (**@**), slash (**/**), and space are not allowed. The *spec_type* specifies whether this is a block or character device. The *minor_num* is the minor number for the device.

The *node_type* is used to create the names in the **/dev** hierarchy that refers to the names in the **/devices** hierarchy. See **disks**(1M), **ports**(1M), **tapes**(1M), **devlinks**(1M). Finally *is_clone* determines if this is a clone device or not.

RETURN VALUES

ddi_create_minor_node() returns:

DDI_SUCCESS if it was able to allocate memory, create the minor data structure, and place it into the linked list of minor devices for this driver.

DDI_FAILURE if minor node creation failed.

EXAMPLES

The following example creates a data structure describing a minor device called **foo** which has a minor number of 0. It is of type **DDI_NT_BLOCK** (a block device) and it is not a clone device.

```
ddi_create_minor_node(dip, "foo", S_IFBLK, 0, DDI_NT_BLOCK, 0);
```

SEE ALSO

add_drv(1M), **devlinks**(1M), **disks**(1M), **drvconfig**(1M), **ports**(1M), **tapes**(1M), **attach**(9E), **ddi_remove_minor_node**(9F)

Writing Device Drivers

NAME

ddi_dev_is_sid – tell whether a device is self-identifying

SYNOPSIS

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

int ddi_dev_is_sid(dev_info_t ∗*dip***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dev_is_sid( ) tells the caller whether the device described by *dip* is self-identifying, that is, a device that can unequivocally tell the system that it exists. This is useful for drivers that support both a self-identifying as well as a non-self-identifying variants of a device (and therefore must be probed).

ARGUMENTS

dip   A pointer to the device's **dev_info** structure.

RETURN VALUES

DDI_SUCCESS   Device is self-identifying.

DDI_FAILURE   Device is not self-identifying.

CONTEXT

ddi_dev_is_sid( ) can be called from user or interrupt context.

EXAMPLE

```
...
int
bz_probe(dev_info_t *dip)
{
        ...
        if (ddi_dev_is_sid(dip) == DDI_SUCCESS) {
                /*
                 * This is the self-identifying version (OpenBoot).
                 * No need to probe for it because we know it is there.
                 * The existence of dip && ddi_dev_is_sid() proves this.
                 */
                        return (DDI_PROBE_DONTCARE);
        }
        /*
         * Not a self-identifying variant of the device. Now we have to
         * do some work to see whether it is really attached to the
         * system.
         */
...
```

SEE ALSO

probe(9E)

Writing Device Drivers

NAME

ddi_dev_nintrs – return the number of interrupt specifications a device has

SYNOPSIS

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

int ddi_dev_nintrs(dev_info_t **dip***, int** **resultp***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dev_nintrs( ) returns the number of interrupt specifications a device has in **resultp*.

RETURN VALUES

ddi_dev_nintrs( ) returns:

DDI_SUCCESS A successful return. The number of interrupt specifications that the device has is set in *resultp*.

DDI_FAILURE The device has no interrupt specifications.

CONTEXT

ddi_dev_nintrs( ) can be called from user or interrupt context.

SEE ALSO

sbus(4), **vme**(4), **isa**(4), **ddi_add_intr**(9F), **ddi_dev_nregs**(9F), **ddi_dev_regsize**(9F)

Writing Device Drivers

NAME

ddi_dev_nregs – return the number of register sets a device has

SYNOPSIS

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

int ddi_dev_nregs(dev_info_t **dip*, **int** **resultp*)

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

The function **ddi_dev_nregs**( ) returns the number of sets of registers the device has.

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

resultp Pointer to an integer that holds the number of register sets on return.

RETURN VALUES

ddi_dev_nregs( ) returns:

DDI_SUCCESS A successful return. The number of register sets is returned in *resultp*.

DDI_FAILURE The device has no registers.

CONTEXT

ddi_dev_nregs( ) can be called from user or interrupt context.

SEE ALSO

ddi_dev_nintrs(9F), **ddi_dev_regsize**(9F)

Writing Device Drivers

NAME

ddi_dev_regsize – return the size of a device's register

SYNOPSIS

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

int ddi_dev_regsize(dev_info_t **dip*, **u_int** *rnumber*, **off_t** **resultp*)

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dev_regsize( ) returns the size, in bytes, of the device register specified by *dip* and *rnumber*. This is useful when, for example, one of the registers is a frame buffer with a varying size known only to its proms.

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

rnumber The ordinal register number. Device registers are associated with a **dev_info** and are enumerated in arbitrary sets from **0** on up. The number of registers a device has can be determined from a call to **ddi_dev_nregs**(9F).

resultp Pointer to an integer that holds the size, in bytes, of the described register (if it exists).

RETURN VALUES

ddi_dev_regsize( ) returns:

DDI_SUCCESS A successful return. The size, in bytes, of the specified register, is set in *resultp*.

DDI_FAILURE An invalid (nonexistent) register number was specified.

CONTEXT

ddi_dev_regsize( ) can be called from user or interrupt context.

SEE ALSO

ddi_dev_nintrs(9F), **ddi_dev_nregs**(9F)

Writing Device Drivers

NAME

ddi_dma_addr_setup – easier DMA setup for use with virtual addresses

SYNOPSIS

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

int ddi_dma_addr_setup(dev_info_t **dip*, **struct as** **as*,
  caddr_t *addr*, **u_int** *len*, **u_int** *flags*,
  int (**waitfp* **)(caddr_t), caddr_t** *arg*,
  ddi_dma_lim_t **lim*, **ddi_dma_handle_t** **handlep***);**

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

as A pointer to an address space structure. Should be set to **NULL**, which implies kernel address space.

addr Virtual address of the memory object.

len Length of the memory object in bytes.

flags Flags that would go into the **ddi_dma_req** structure (see **ddi_dma_req**(9S)).

waitfp The address of a function to call back later if resources aren't available now. The special function addresses **DDI_DMA_SLEEP** and **DDI_DMA_DONTWAIT** (see **ddi_dma_req**(9S)) are taken to mean, respectively, wait until resources are available or, do not wait at all and do not schedule a callback.

arg Argument to be passed to a callback function, if such a function is specified.

lim A pointer to a DMA limits structure for this device (see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). If this pointer is **NULL**, a default set of DMA limits is assumed.

handlep Pointer to a DMA handle. See **ddi_dma_setup**(9F) for a discussion of handle.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_addr_setup( ) is an interface to **ddi_dma_setup**(9F). It uses its arguments to construct an appropriate **ddi_dma_req** structure and calls **ddi_dma_setup( )** with it.

RETURN VALUES

See **ddi_dma_setup**(9F) for the possible return values for this function.

CONTEXT

ddi_dma_addr_setup( ) can be called from user or interrupt context, except when *waitfp* is set to **DDI_DMA_SLEEP**, in which case it can be called from user context only.

SEE ALSO

ddi_dma_buf_setup(9F), **ddi_dma_free**(9F), **ddi_dma_htoc**(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **ddi_iopb_alloc**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_buf_setup – easier DMA setup for use with buffer structures

SYNOPSIS

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

int ddi_dma_buf_setup(dev_info_t **dip*, **struct buf** **bp*,
 u_int *flags*, **int (****waitfp***)(caddr_t), caddr_t** *arg*,
 ddi_dma_lim_t **lim*, **ddi_dma_handle_t** **handlep***);**

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

bp A pointer to a system buffer structure (see **buf**(9S)).

flags Flags that go into a **ddi_dma_req** structure (see **ddi_dma_req**(9S)).

waitfp The address of a function to call back later if resources aren't available now. The special function addresses **DDI_DMA_SLEEP** and **DDI_DMA_DONTWAIT** (see **ddi_dma_req**(9S)) are taken to mean, respectively, wait until resources are available, or do not wait at all and do not schedule a callback.

arg Argument to be passed to a callback function, if such a function is specified.

lim A pointer to a DMA limits structure for this device (see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). If this pointer is **NULL,** a default set of DMA limits is assumed.

handlep Pointer to a DMA handle. See **ddi_dma_setup**(9F) for a discussion of handle.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_buf_setup( ) is an interface to **ddi_dma_setup**(9F). It uses its arguments to construct an appropriate **ddi_dma_req** structure and calls **ddi_dma_setup**( ) with it.

RETURN VALUES

See **ddi_dma_setup**(9F) for the possible return values for this function.

CONTEXT

ddi_dma_buf_setup( ) can be called from user or interrupt context, except when *waitfp* is set to **DDI_DMA_SLEEP,** in which case it can be called from user context only.

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_free**(9F), **ddi_dma_htoc**(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **physio**(9F), **buf**(9S), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_burstsizes – find out the allowed burst sizes for a DMA mapping

SYNOPSIS

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

int ddi_dma_burstsizes(ddi_dma_handle_t *handle***)**

ARGUMENTS

handle A DMA *handle* that was filled in by a successful call to **ddi_dma_setup**(9F).

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_burstsizes( ) returns the allowed burst sizes for a DMA mapping. This value is derived from the **dlim_burstsizes** member of the **ddi_dma_lim_sparc**(9S) structure, but it shows the allowable burstsizes *after* imposing on it the limitations of other device layers in addition to device's own limitations.

RETURN VALUES

ddi_dma_burstsizes( ) returns a binary encoded value of the allowable DMA burst sizes. See **ddi_dma_lim_sparc**(9S) for a discussion of DMA burst sizes.

CONTEXT

This function can be called from user or interrupt context.

SEE ALSO

ddi_dma_devalign(9F), **ddi_dma_setup**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_coff – convert a DMA cookie to an offset within a DMA handle

SYNOPSIS

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

int ddi_dma_coff(ddi_dma_handle_t *handle*, **ddi_dma_cookie_t** **cookiep*, **off_t** **offp***)**

ARGUMENTS

handle — The *handle* filled in by a call to **ddi_dma_setup**(9F).

cookiep — A pointer to a DMA cookie (see **ddi_dma_cookie**(9S)) that contains the appropriate address, length and bus type to be used in programming the DMA engine.

offp — A pointer to an offset to be filled in.

INTERFACE LEVEL

Solaris SPARC DDI (Solaris SPARC DDI).

DESCRIPTION

ddi_dma_coff( ) converts the values in DMA cookie pointed to by *cookiep* to an offset (in bytes) from the beginning of the object that the DMA **handle** has mapped.

ddi_dma_coff( ) allows a driver to update a DMA cookie with values it reads from its device's DMA engine after a transfer completes and convert that value into an offset into the object that is mapped for DMA.

RETURN VALUES

ddi_dma_coff( ) returns:

DDI_SUCCESS — Successfully filled in *offp*.

DDI_FAILURE — Failed to successfully fill in *offp*.

CONTEXT

ddi_dma_coff( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_setup(9F), **ddi_dma_sync**(9F), **ddi_dma_cookie**(9S)

Writing Device Drivers

NAME

ddi_dma_curwin – report current DMA window offset and size

SYNOPSIS

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

int ddi_dma_curwin(ddi_dma_handle_t *handle***, off_t** **offp***,u_int** **lenp***);**

ARGUMENTS

handle The DMA handle filled in by a call to **ddi_dma_setup**(9F).

offp A pointer to a value which will be filled in with the current offset from the beginning of the object that is mapped for DMA.

lenp A pointer to a value which will be filled in with the size, in bytes, of the current window onto the object that is mapped for DMA.

INTERFACE LEVEL

Solaris SPARC DDI specific (Solaris SPARC DDI).

DESCRIPTION

ddi_dma_curwin( ) reports the current DMA window offset and size. If a DMA mapping allows partial mapping, that is if the **DDI_DMA_PARTIAL** flag in the **ddi_dma_req**(9S) structure is set, its current (effective) DMA window offset and size can be obtained by a call to **ddi_dma_curwin( )**.

RETURN VALUES

ddi_dma_curwin( ) returns:

DDI_SUCCESS The current length and offset can be established.

DDI_FAILURE Otherwise.

CONTEXT

ddi_dma_curwin( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_movwin(9F), **ddi_dma_setup**(9F), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_devalign – find DMA mapping alignment and minimum transfer size

SYNOPSIS

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

int ddi_dma_devalign(ddi_dma_handle_t *handle*, **u_int** **alignment*, **u_int** **minxfr***);**

ARGUMENTS

handle The DMA **handle** filled in by a successful call to **ddi_dma_setup**(9F).

alignment A pointer to an unsigned integer to be filled in with the minimum required alignment for DMA. The alignment is guaranteed to be a power of two.

minxfr A pointer to an unsigned integer to be filled in with the minimum effective transfer size (see **ddi_iomin**(9F), **ddi_dma_lim_sparc**(9S) and **ddi_dma_lim_x86**(9S)). This also is guaranteed to be a power of two.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_devalign( ) determines (after a successful DMA mapping (see **ddi_dma_setup**(9F)) the minimum required data alignment and minimum DMA transfer size.

RETURN VALUES

ddi_dma_devalign( ) returns:

DDI_SUCCESS The *alignment* and *minxfr* values have been filled.

DDI_FAILURE The handle was illegal.

CONTEXT

ddi_dma_devalign( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_setup(9F), **ddi_iomin**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_free – release system DMA resources

SYNOPSIS

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

int ddi_dma_free(ddi_dma_handle_t *handle***);**

ARGUMENTS

handle The handle filled in by a call to **ddi_dma_setup**(9F).

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_free( ) releases system DMA resources set up by **ddi_dma_setup**(9F). When a DMA transfer completes, the driver should free up system DMA resources established by a call to **ddi_dma_setup**(9F). This is done by a call to **ddi_dma_free**( ). **ddi_dma_free**( ) does an implicit **ddi_dma_sync**(9F) for you so any further synchronization steps are not necessary.

RETURN VALUES

ddi_dma_free( ) returns:

DDI_SUCCESS Successfully released resources

DDI_FAILURE Failed to free resources

CONTEXT

ddi_dma_free( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_buf_setup**(9F), **ddi_dma_htoc**(9F), **ddi_dma_sync**(9F), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_htoc – convert a DMA handle to a DMA address cookie

SYNOPSIS

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

int ddi_dma_htoc(ddi_dma_handle_t *handle*, **off_t** *off*, **ddi_dma_cookie_t** **cookiep***);**

ARGUMENTS

handle The handle filled in by a call to **ddi_dma_setup**(9F).

off An offset into the object that *handle* maps.

cookiep A pointer to a **ddi_dma_cookie**(9S) structure.

INTERFACE LEVEL

Solaris SPARC DDI specific (Solaris SPARC DDI).

DESCRIPTION

ddi_dma_htoc( ) takes a DMA handle (established by **ddi_dma_setup**(9F)), and fills in the cookie pointed to by *cookiep* with the appropriate address, length, and bus type to be used to program the DMA engine.

RETURN VALUES

ddi_dma_htoc( ) returns:

DDI_SUCCESS Successfully filled in the cookie pointed to by *cookiep*.

DDI_FAILURE Failed to successfully fill in the cookie.

CONTEXT

ddi_dma_htoc( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_buf_setup**(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **ddi_dma_cookie**(9S)

Writing Device Drivers

NAME

ddi_dma_movwin – shift current DMA window

SYNOPSIS

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

int ddi_dma_movwin(ddi_dma_handle_t *handle***, off_t** **offp***, u_int** **lenp***,**
ddi_dma_cookie_t **cookiep***);**

ARGUMENTS

handle — The DMA handle filled in by a call to **ddi_dma_setup**(9F).

offp — A pointer to an offset to set the DMA window to. Upon a successful return, it will be filled in with the new offset from the beginning of the object resources are allocated for.

lenp — A pointer to a value which must either be the current size of the DMA window (as known from a call to **ddi_dma_curwin**(9F) or from a previous call to **ddi_dma_movwin()**). Upon a successful return, it will be filled in with the size, in bytes, of the current window.

cookiep — A pointer to a DMA cookie (see **ddi_dma_cookie**(9S)). Upon a successful return, cookiep is filled in just as if an implicit **ddi_dma_htoc**(9F) had been made.

INTERFACE LEVEL

Solaris SPARC DDI specific (Solaris SPARC DDI).

DESCRIPTION

ddi_dma_movwin( ) shifts the current DMA window. If a DMA request allows the sytem to allocate resources for less than the entire object by setting the **DDI_DMA_PARTIAL** flag in the **ddi_dma_req**(9S) structure, the current DMA window can be shifted by a call to **ddi_dma_movwin**( ).

The caller must first determine the current DMA window size by a call to **ddi_dma_curwin**(9F). Using the current offset and size of the window thus retrieved, the caller of **ddi_dma_movwin( )** may change the window onto the object by changing the offset by a value which is some multiple of the size of the DMA window.

ddi_dma_movwin( ) takes care of underlying resource synchronizations required to *shift* the window. However if you want to *access* the data prior or after moving the window, further synchronizations using **ddi_dma_sync**(9F) are required,

This function is normally called from an interrupt routine. The first invocation of the DMA engine is done from the driver. All subsequent invocations of the DMA engine are done from the interrupt routine. The interrupt routine checks to see if the request has been completed. If it has, it returns without invoking another DMA transfer. Otherwise it calls **ddi_dma_movwin( )** to shift the current window and starts another DMA transfer.

RETURN VALUES

ddi_dma_movwin( ) returns:

DDI_SUCCESS — The current length and offset are legal and have been set.

DDI_FAILURE — Otherwise.

CONTEXT

ddi_dma_movwin( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_curwin(9F), **ddi_dma_htoc**(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **ddi_dma_cookie**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

WARNINGS

The caller must guarantee that the resources used by the object are inactive prior to calling this function.

NAME

ddi_dma_nextseg – get next DMA segment

SYNOPSIS

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

int ddi_dma_nextseg(ddi_dma_win_t *win*, **ddi_dma_seg_t** *seg*, **ddi_dma_seg_t** **nseg***);**

ARGUMENTS

win	A DMA *window*.
seg	The current DMA segment or **NULL**.
nseg	A pointer to the next DMA segment to be filled in. If *seg* is **NULL**, a pointer to the first segment within the specified window is returned.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_nextseg( ) gets the next DMA segment within the specified window *win*. If the current segment is **NULL**, the first DMA segment within the window is returned.

A DMA segment is always required for a DMA window. A DMA segment is a contiguous portion of a DMA window (see **ddi_dma_nextwin**(9F)) which is entirely addressable by the device for a data transfer operation.

An example where multiple DMA segments are allocated is where the system does not contain DVMA capabilities and the object may be non-contiguous. In this example the object will be broken into smaller contiguous DMA segments. Another example is where the device has an upper limit on its transfer size (for example an 8-bit address register) and has expressed this in the DMA limit structure (see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). In this example the object will be broken into smaller addressable DMA segments.

RETURN VALUES

ddi_dma_nextseg( ) returns:

DDI_SUCCESS	Successfully filled in the next segment pointer.
DDI_DMA_DONE	There is no next segment. The current segment is the final segment within the specified window.
DDI_DMA_STALE	*win* does not refer to the currently active window.

CONTEXT

ddi_dma_nextseg( ) can be called from user or interrupt context.

EXAMPLE

For an example see **ddi_dma_segtocookie**(9F).

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_buf_setup**(9F), **ddi_dma_nextwin**(9F), **ddi_dma_req**(9S), **ddi_dma_segtocookie**(9F), **ddi_dma_sync**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S)

Writing Device Drivers

NAME

ddi_dma_nextwin – get next DMA window

SYNOPSIS

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

int ddi_dma_nextwin(ddi_dma_handle_t *handle,* **ddi_dma_win_t** *win,*
ddi_dma_win_t **nwin***);**

ARGUMENTS

handle — A DMA *handle.*

win — The current DMA window or **NULL**.

nwin — A pointer to the next DMA window to be filled in. If *win* is **NULL**, a pointer to the first window within the object is returned.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_nextwin( ) shifts the current DMA window *win* within the object referred to by *handle* to the next DMA window *nwin*. If the current window is **NULL**, the first window within the object is returned. A DMA window is a portion of a DMA object or might be the entire object. A DMA window has system resources allocated to it and is prepared to accept data transfers. Examples of system resources are DVMA mapping resources and intermediate transfer buffer resources.

All DMA objects require a window. If the DMA window represents the whole DMA object it has system resources allocated for the entire data transfer. However, if the system is unable to setup the entire DMA object due to system resource limitations, the driver writer may allow the system to allocate system resources for less than the entire DMA object. This can be accomplished by specifying the **DDI_DMA_PARTIAL** flag as a parameter to **ddi_dma_buf_setup**(9F) or **ddi_dma_addr_setup**(9F) or as part of a **ddi_dma_req**(9S) structure in a call to **ddi_dma_setup**(9F).

Only the window that has resources allocated is valid per object at any one time. The currently valid window is the one that was most recently returned from **ddi_dma_nextwin( )**. Furthermore, because a call to **ddi_dma_nextwin( )** will reallocate system resources to the new window, the previous window will become invalid. **Note:** It is a *severe* error to call **ddi_dma_nextwin( )** before any transfers into the current window are complete.

ddi_dma_nextwin( ) takes care of underlying memory synchronizations required to shift the window. However, if you want to access the data before or after moving the window, further synchronizations using **ddi_dma_sync**(9F) are required.

RETURN VALUES

ddi_dma_nextwin( ) returns:

DDI_SUCCESS — Successfully filled in the next window pointer.

DDI_DMA_DONE — There is no next window. The current window is the final window within the specified object.

DDI_DMA_STALE — *win* does not refer to the currently active window.

CONTEXT

ddi_dma_nextwin( ) can be called from user or interrupt context.

EXAMPLE

For an example see **ddi_dma_segtocookie**(9F).

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_buf_setup**(9F), **ddi_dma_nextseg**(9F), **ddi_dma_segtocookie**(9F), **ddi_dma_sync**(9F), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_dma_segtocookie – convert a DMA segment to a DMA address cookie

SYNOPSIS

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

int ddi_dma_segtocookie(ddi_dma_seg_t *seg,* **off_t** *∗offp,* **off_t** *∗lenp,*
ddi_dma_cookie_t *∗cookiep***);**

ARGUMENTS

seg A DMA *segment*.

offp A pointer to an *off_t*. Upon a successful return, it is filled in with the offset. This segment is addressing within the object.

lenp The byte length. This segment is addressing within the object.

cookiep A pointer to a DMA *cookie* (see **ddi_dma_cookie**(9S)).

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_segtocookie( ) takes a DMA segment and fills in the cookie pointed to by *cookiep* with the appropriate address, length, and bus type to be used to program the DMA engine. **ddi_dma_segtocookie( )** also fills in *∗offp* and *∗lenp*, which specify the range within the object.

RETURN VALUES

ddi_dma_segtocookie( ) returns:

DDI_SUCCESS Successfully filled in all values.

DDI_FAILURE Failed to successfully fill in all values.

CONTEXT

ddi_dma_segtocookie( ) can be called from user or interrupt context.

EXAMPLE

```
for (win = NULL; (retw = ddi_dma_nextwin(handle, win, &nwin)) !=
    DDI_DMA_DONE; win = nwin) {
        if (retw != DDI_SUCCESS) {

                /* do error handling */
        } else {
                for (seg = NULL; (rets = ddi_dma_nextseg(nwin, seg, &nseg)) !=
                    DDI_DMA_DONE; seg = nseg) {
                        if (rets != DDI_SUCCESS) {

                                /* do error handling */
                        } else {
                                ddi_dma_segtocookie(nseg, &off, &len, &cookie);

                                /* program DMA engine */
                        }
                }
```

```
                }
        }
```

SEE ALSO

ddi_dma_nextseg(9F), **ddi_dma_nextwin**(9F), **ddi_dma_sync**(9F), **ddi_dma_cookie**(9S)
Writing Device Drivers

NAME

ddi_dma_setup – setup DMA resources

SYNOPSIS

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

int ddi_dma_setup(dev_info_t **dip*, **ddi_dma_req_t** **dmareqp*,
    ddi_dma_handle_t **handlep***);**

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

dmareqp A pointer to a DMA request structure (see **ddi_dma_req**(9S)).

handlep A pointer to a DMA handle to be filled in. See below for a discussion of a handle. If *handlep* is **NULL,** the call to **ddi_dma_setup**( ) is considered an advisory call, in which case no resources are allocated, but a value indicating the legality and the feasibility of the request is returned.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_setup( ) allocates resources for a memory object such that a device can perform DMA to or from that object.

A call to **ddi_dma_setup**( ) informs the system that device referred to by *dip* wishes to perform DMA to or from a memory object. The memory object, the device's DMA capabilities, the device driver's policy on whether to wait for resources, are all specified in the **ddi_dma_req** structure pointed to by *dmareqp*.

A successful call to **ddi_dma_setup**( ) fills in the value pointed to by *handlep*. This is an opaque object called a DMA handle. This handle is then used in subsequent DMA calls, until **ddi_dma_free**(9F) is called.

Again a DMA handle is opaque—drivers may **not** attempt to interpret its value. When a driver wants to enable its DMA engine, it must retrieve the appropriate address to supply to its DMA engine using a call to **ddi_dma_htoc**(9F), which takes a pointer to a DMA handle and returns the appropriate DMA address.

When DMA transfer completes, the driver should free up the the allocated DMA resources by calling **ddi_dma_free**( ).

RETURN VALUES

ddi_dma_setup( ) returns:

DDI_DMA_MAPPED Successfully allocated resources for the object. In the case of an *advisory* call, this indicates that the request is legal.

DDI_DMA_PARTIAL_MAP Successfully allocated resources for a *part* of the object. This is acceptable when partial transfers are allowed using a flag setting in the **ddi_dma_req** structure (see **ddi_dma_req**(9S) and **ddi_dma_movwin**(9F)).

DDI_DMA_NORESOURCES When no resources are available.

DDI_DMA_NOMAPPING	The object cannot be reached by the device requesting the resources.
DDI_DMA_TOOBIG	The object is too big and exceeds the available resources. The maximum size varies depending on machine and configuration.

CONTEXT

ddi_dma_setup( ) can be called from user or interrupt context, except when the **dmar_fp** member of the **ddi_dma_req** structure pointed to by *dmareqp* is set to **DDI_DMA_SLEEP,** in which case it can be called from user context only.

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_buf_setup**(9F), **ddi_dma_free**(9F), **ddi_dma_htoc**(9F), **ddi_dma_sync**(9F), **ddi_dma_req**(9S)

Writing Device Drivers

NOTES

The construction of the **ddi_dma_req** structure is complicated. Use of the provided interface functions such as **ddi_dma_buf_setup**(9F) simplifies this task.

NAME

ddi_dma_sync – synchronize CPU and I/O views of memory

SYNOPSIS

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

int ddi_dma_sync(ddi_dma_handle_t *handle***, off_t** *offset***, u_int** *length***, u_int** *type***);**

ARGUMENTS

handle The *handle* filled in by a call to **ddi_dma_setup**(9F).

offset The offset into the object described by the *handle*.

length The length, in bytes, of the area to synchronize. When *length* is zero, the entire range starting from **offset** to the end of the object has the requested operation applied to it.

type Indicates the caller's desire about what view of the memory object to synchronize. The possible values are **DDI_DMA_SYNC_FORDEV**, **DDI_DMA_SYNC_FORCPU** and **DDI_DMA_SYNC_FORKERNEL**.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_dma_sync( ) is used to selectively synchronize either a DMA device's or a CPU's view of a memory object that has been mapped for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When a memory object is mapped for DMA , you may assume that an implicit **ddi_dma_sync( )** is done for you when you call **ddi_dma_setup( )**. When a memory object is unmapped via a call to **ddi_dma_free**(9F), you may assume that an implicit **ddi_dma_sync( )** is done for you. However, at any time between mapping a memory object for DMA and unmapping it after DMA completes, if the memory object has been modified by either the DMA device or a CPU and you wish to ensure that the change is noticed by the party that *didn't* do the modifying, a call to **ddi_dma_sync( )** is required. This is true *independent* of any attributes of the memory object including, but not limited to, whether or not the memory was allocated for non-streaming mode I/O (see **ddi_iopb_alloc**(9F)) or whether or not the memory was mapped for DMA in non-streaming mode (see **ddi_dma_req**(9S)).

This cannot be stated too strongly. If a consistent view of the memory object must be ensured between the time you map the object for DMA and the time you free such a mapping, you **must** call **ddi_dma_sync( )** to ensure that either a CPU or a DMA device has such a consistent view.

What to set *type* to depends on the view you are trying to ensure consistency for. If the memory object is modified by a CPU , and the object is going to be *read* by the DMA engine of your device, you use **DDI_DMA_SYNC_FORDEV**. This ensures that your device's DMA engine sees any changes that a CPU has made to the memory object. If the DMA engine for your device has *written* to the memory object, and you are going to *read* (with a CPU) the object (using an extant virtual address mapping that you have to the memory object), you use **DDI_DMA_SYNC_FORCPU**. This ensures that a CPU's view of the memory object includes any changes made to the object by your device's DMA engine. If you are only interested in the kernel's view (kernel-space part of the CPU's view) you may use **DDI_DMA_SYNC_FORKERNEL**. This gives a *hint* to the system—that is, if it is more economical to synchronize the kernel's view only, then do so; otherwise, synchronize for CPU.

RETURN VALUES

ddi_dma_sync( ) returns:

DDI_SUCCESS Caches are successfully flushed.

DDI_FAILURE The address range to be flushed is out of the address range established by **ddi_dma_setup**(9F).

CONTEXT

ddi_dma_sync( ) can be called from user or interrupt context.

SEE ALSO

ddi_dma_free(9F), **ddi_dma_setup**(9F), **ddi_iopb_alloc**(9F)

Writing Device Drivers

NAME

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

SYNOPSIS

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

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

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

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

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

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

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

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

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

AVAILABILITY

x86

INTERFACE LEVEL

Solaris x86 DDI specific (Solaris x86 DDI).

ARGUMENTS

dip A **dev_info** pointer, which identifies the device.

chnl A DMA channel number, or an MCA bus arbitration level. On ISA or EISA buses this number must be **0**, **1**, **2**, **3**, **5**, **6**, or **7**. On MCA buses this number must be in the range **0** to **14**.

dmae_waitfp A wait / callback function address.

arg The argument to be passed to the callback function.

dmaereqp A pointer to a DMA engine request (**ddi_dmae_req**(9S)) structure.

cookiep A pointer to a **ddi_dma_cookie**(9S) object, obtained from **ddi_dma_segtocookie**(9F), which contains the address and count.

countp A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.

limitsp A pointer to a DMA limit structure. See **ddi_dma_lim_x86**(9S).

DESCRIPTION

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

Bus master DMA If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from **ddi_dma_segtocookie( )**. See **ddi_dma_cookie**(9S) for a description of a DMA cookie.

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

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

ddi_dmae_alloc( )

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

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

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

ddi_dmae_prog( )

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

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

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

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

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

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

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

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

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

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

RETURN VALUES

DDI_SUCCESS Upon success, for all of these routines.

DDI_FAILURE May be returned due to invalid arguments.

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

CONTEXT

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

SEE ALSO

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

NAME

ddi_enter_critical, ddi_exit_critical – enter and exit a critical region of control

SYNOPSIS

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

unsigned int ddi_enter_critical(void)
void ddi_exit_critical(unsigned int *ddic***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ddic The returned value from the call to **ddi_enter_critical**( ) must be passed to **dd_exit_critical**( ).

DESCRIPTION

Nearly all driver operations can be done without any special synchronization and protection mechanisms beyond those provided by, e.g., *mutexes* (see **mutex**(9F)). However, for certain devices there can exist a very short critical region of code which *must* be allowed to run uninterrupted. The function **ddi_enter_critical**( ) provides a mechanism by which a driver can ask the system to guarantee to the best of its ability that the current thread of execution will neither be preempted nor interrupted. This stays in effect until a bracketing call to **ddi_exit_critical**( ) is made (with an argument which was the returned value from **ddi_enter_critical**( )).

The driver may not call any functions external to itself in between the time it calls **ddi_enter_critical**( ) and the time it calls **ddi_exit_critical**( ).

RETURN VALUES

ddi_enter_critical( ) returns an opaque unsigned integer which must be used in the subsequent call to **ddi_exit_critical**( ).

CONTEXT

This function can be called from user or interrupt context.

WARNINGS

Driver writers should note that in a multiple processor system this function does not temporarily suspend other processors from executing. This function also cannot guarantee to actually block the hardware from doing such things as interrupt acknowledge cycles. What it *can* do is guarantee that the currently executing thread will not be preempted.

Do not write code bracketed by **ddi_enter_critical**( ) and **ddi_exit_critical**( ) that can get caught in an infinite loop, as the machine may crash if you do.

SEE ALSO

mutex(9F)

Writing Device Drivers

NAME

ddi_ffs, ddi_fls – find first (last) bit set in a long integer

SYNOPSIS

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

int ddi_ffs(long *mask***)**
int ddi_fls(long *mask***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

mask A 32 bit argument value to search through.

DESCRIPTION

The function **ddi_ffs**( ) takes its argument and returns the shift count that the first (least significant) bit set in the argument corresponds to. The function **ddi_fls**( ) does the same, only it returns the shift count for the last (most significant) bit set in the argument.

RETURN VALUES

N Returns a number from 1 to 31 which corresponds to the bit position of either the least significant (first) or most significant (last) bit set in the argument.

CONTEXT

This function can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

ddi_get_cred – returns a pointer to the credential structure of the caller.

SYNOPSIS

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

cred_t *ddi_get_cred( );

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_get_cred( ) returns a pointer to the user credential structure of the caller.

RETURN VALUES

ddi_get_cred( ) returns a pointer to the caller's credential structure.

CONTEXT

ddi_get_cred( ) can be called from user context only.

SEE ALSO

Writing Device Drivers

NAME

ddi_get_driver_private, ddi_set_driver_private – get or set the address of the device's private data area

SYNOPSIS

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

void ddi_set_driver_private(dev_info_t **dip*, **caddr_t** *data***)**
caddr_t ddi_get_driver_private(dev_info_t **dip***)**

ARGUMENTS

ddi_get_driver_private( )

dip Pointer to device information structure to get from.

ddi_set_driver_private( )

dip Pointer to device information structure to set.

data Data area address to set.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_get_driver_private( ) returns the address of the device's private data area from the device information structure pointed to by *dip*.

ddi_set_driver_private( ) sets the address of the device's private data area in the device information structure pointed to by *dip* with the value of *data*.

RETURN VALUES

ddi_get_driver_private( ) returns the address of the private data area. If **ddi_set_driver_private**( ) has not been previously called with *dip*, an unpredictable value is returned.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

ddi_get_instance – get device instance number

SYNOPSIS

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

int ddi_get_instance(dev_info_t **dip***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip Pointer to **dev_info** structure.

DESCRIPTION

ddi_get_instance( ) returns the instance number of the device corresponding to *dip*. Instance number ranges from zero to the number of devices attached to the driver minus one.

RETURN VALUES

ddi_get_instance( ) returns an integer between **0** and the number of instances of this device.

CONTEXT

ddi_get_instance( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

ddi_get_name – return the devinfo node name

SYNOPSIS

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

char ∗**ddi_get_name(dev_info_t** ∗*dip***);**

ARGUMENTS

dip A pointer the device's **dev_info** structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_get_name( ) returns the name contained in the **dev_info** node pointed to by *dip*.

RETURN VALUES

ddi_get_name( ) returns the name contained in the **dev_info** structure.

CONTEXT

ddi_get_name( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME ddi_get_parent – find the parent of a device information structure

SYNOPSIS

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

dev_info_t *ddi_get_parent(dev_info_t * *dip* **);**

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip Pointer to a device information structure.

DESCRIPTION **ddi_get_parent**( ) returns a pointer to the device information structure which is the parent of the one pointed to by *dip*.

RETURN VALUES **ddi_get_parent**( ) returns a pointer to a device information structure.

CONTEXT **ddi_get_parent**( ) can be called from user or interrupt context.

SEE ALSO *Writing Device Drivers*

NAME

ddi_intr_hilevel – indicate interrupt handler type

SYNOPSIS

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

int ddi_intr_hilevel(dev_info_t **dip*, **u_int** *inumber***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip Pointer to **dev_info** structure.

inumber Interrupt number.

DESCRIPTION

ddi_intr_hilevel( ) returns non-zero if the specified interrupt is a "high level" interrupt.

High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler.

In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable.

A typical high level interrupt handler would put data into a circular buffer and schedule a soft interrupt by calling **ddi_trigger_softintr**( ). The circular buffer could be protected by using a mutex that was properly initialized for the interrupt handler.

ddi_intr_hilevel( ) can be used before calling **ddi_add_intr**( ) to decide which type of interrupt handler should be used. Most device drivers are designed with the knowledge that the devices they support will always generate low level interrupts, however some devices, for example those using S-bus or VME bus level 6 or 7 interrupts must use this test because on some machines those interrupts are high level (above the scheduler level) and on other machines they are not.

RETURN VALUES

non-zero indicates a high-level interrupt.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

ddi_add_intr(9F), **mutex**(9F)

Writing Device Drivers

NAME

ddi_iomin – find minimum alignment and transfer size for DMA

SYNOPSIS

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

int ddi_iomin(dev_info_t **dip*, **int** *initial*, **int** *streaming***)**

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

initial The initial minimum DMA transfer size in bytes. This may be zero or an appropriate **dlim_minxfer** value for device's **ddi_dma_lim** structure (see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). This value must be a power of two.

streaming This argument, if non-zero, indicates that the returned value should be modified to account for *streaming* mode accesses (see **ddi_dma_req**(9S) for a discussion of streaming versus non-streaming access mode).

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_iomin( ), finds out the minimum DMA transfer size for the device pointed to by *dip*. This provides a mechanism by which a driver can determine the effects of underlying caches as well as intervening bus adapters on the granularity of a DMA transfer.

RETURN VALUES

ddi_iomin( ) returns the minimum DMA transfer size for the calling device, or it returns zero, which means that you cannot get there from here.

CONTEXT

This function can be called from user or interrupt context.

SEE ALSO

ddi_dma_devalign(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_iopb_alloc, ddi_iopb_free – allocate and free non-sequentially accessed memory

SYNOPSIS

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

int ddi_iopb_alloc(dev_info_t **dip,* **ddi_dma_lim_t** **limits,* **u_int** *length*,
caddr_t **iopbp***);**

void ddi_iopb_free(caddr_t *iopb***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ddi_iopb_alloc( )

dip A pointer to the device's **dev_info** structure.

limits A pointer to a DMA limits structure for this device (see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). If this pointer is **NULL**, a default set of DMA limits is assumed.

length The length in bytes of the desired allocation.

iopbp A pointer to a **caddr_t**. On a successful return, **iopbp* points to the allocated storage.

ddi_iopb_free( )

iopb The *iopb* returned from a successful call to **ddi_iopb_alloc( )**.

DESCRIPTION

ddi_iopb_alloc( ) allocates memory for DMA transfers and should be used if the device accesses memory in a non-sequential fashion, or if synchronization steps using **ddi_dma_sync**(9F) should be as lightweight as possible, due to frequent use on small objects. This type of access is commonly known as *consistent* access. The allocation will obey the alignment and padding constraints as specified in the *limits* argument and other limits imposed by the system.

Note that you still must use DMA resource allocation functions (see **ddi_dma_setup**(9F)) to establish DMA resources for the memory allocated using **ddi_iopb_alloc( )**.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using **ddi_dma_sync**(9F) or **ddi_dma_free**(9F) are still required. The DMA resources will be allocated so that these synchronization steps are as efficient as possible.

ddi_iopb_free( ) frees up memory allocated by **ddi_iopb_alloc( )**.

RETURN VALUES

ddi_iopb_alloc( ) returns:

DDI_SUCCESS Memory successfully allocated.

DDI_FAILURE Allocation failed.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

ddi_dma_free(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **ddi_iopb_free**(9F), **ddi_mem_alloc**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NOTES

This function uses scarce system resources. Use it selectively.

NAME

ddi_map_regs, ddi_unmap_regs – map or unmap registers

SYNOPSIS

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

int ddi_map_regs(dev_info_t **dip*, **u_int** *rnumber*, **caddr_t** **kaddrp,*
off_t *offset*, **off_t** *len*);

void ddi_unmap_regs(dev_info_t **dip*, **u_int** *rnumber*, **caddr_t** **kaddrp,*
off_t *offset*, **off_t** *len*);

ARGUMENTS

ddi_map_regs()

dip	Pointer to the device's dev_info structure.
rnumber	Register set number.
kaddrp	Pointer to the base kernel address of the mapped region (set on return).
offset	Offset into register space.
len	Length to be mapped.

ddi_unmap_regs()

dip	Pointer to the device's dev_info structure.
rnumber	Register set number.
kaddrp	Pointer to the base kernel address of the region to be unmapped.
offset	Offset into register space.
len	Length to be unmapped.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_map_regs() maps in the register set given by *rnumber*. The register number determines which register set will be mapped if more than one exists. The base kernel virtual address of the mapped register set is returned in *kaddrp*. *offset* specifies an offset into the register space to start from and *len* indicates the size of the area to be mapped. If *len* is non-zero, it overrides the length given in the register set description. See the discussion of the **reg** property in **sbus**(4) and **vme**(4) for more information on register set descriptions. If *len* and *offset* are 0, the entire space is mapped.

ddi_unmap_regs() undoes mappings set up by **ddi_map_regs()**. This is provided for drivers preparing to detach themselves from the system, allowing them to release allocated mappings. Mappings must be released in the same way they were mapped (a call to **ddi_unmap_regs()** must correspond to a previous call to **ddi_map_regs()).** Releasing portions of previous mappings is not allowed. *rnumber* determines which register set will be unmapped if more than one exists. The *kaddrp*, *offset* and *len* specify the area to be unmapped. *kaddrp* is a pointer to the address returned from **ddi_map_regs()**; *offset* and *len* should match what **ddi_map_regs()** was called with.

RETURN VALUES

ddi_map_regs() returns:

DDI_SUCCESS on success.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

sbus(4), **vme**(4)

Writing Device Drivers

NAME

ddi_mapdev – create driver-controlled mapping of device

SYNOPSIS

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

int ddi_mapdev(dev_t *dev,* **off_t** *offset,* **struct as** **as,* **caddr_t** **addrp,* **off_t** *len,* **u_int** *prot,* **u_int** *maxprot,* **u_int** *flags,* **cred_t** **cred,* **struct ddi_mapdev_ctl** **ctl,* **ddi_mapdev_handle_t** **handlep,* **void** **devprivate***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dev — The device whose memory is to be mapped.

offset — The offset within device memory at which the mapping begins.

as — An opaque pointer to the user address space into which the device memory should be mapped.

addrp — Pointer to the starting address within the user address space to which the device memory should be mapped.

len — Length (in bytes) of the memory to be mapped.

prot — A bit field that specifies the protections. Some combinations of possible settings are:

- **PROT_READ** — Read access is desired.
- **PROT_WRITE** — Write access is desired.
- **PROT_EXEC** — Execute access is desired.
- **PROT_USER** — User-level access is desired (the mapping is being done as a result of a **mmap**(2) system call).
- **PROT_ALL** — All access is desired.

maxprot — Maximum protection flag possible for attempted mapping (the **PROT_WRITE** bit may be masked out if the user opened the special file read-only). If **(maxprot & prot) != prot** then there is an access violation.

flags — Flags indicating type of mapping. Possible values are (other bits may be set):

- **MAP_PRIVATE** — Changes are private.

cred — Pointer to the user credentials structure.

ctl — A pointer to a **ddi_mapdev_ctl**(9S) structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.

handlep — An opaque pointer to a device mapping handle. A handle to the new device mapping is generated and placed into the location pointed to by **handlep*. If the call fails, the value of **handlep* is undefined.

devprivate — Driver private mapping data. This value is passed into each mapping call back routine.

DESCRIPTION

ddi_mapdev() sets up user mappings to device space in the same manner as **ddi_segmap**(9F). However, unlike mappings created with **ddi_segmap()**, mappings created with **ddi_mapdev()** have a set of driver entry points and a mapping handle associated with them. The driver is notified via these entry points in response to user events on the mappings. The events defined on these mappings are:

access	User has accessed an address in the mapping that has no translations.
duplication	User has duplicated the mapping. Mappings are duplicated when the process calls **fork**(2).
unmapping	User has called **munmap**(2) on the mapping or is exiting.

See **mapdev_access**(9E), **mapdev_dup**(9E), and **mapdev_free**(9E) for details on these entry points.

With the handle, device drivers can use **ddi_mapdev_intercept**(9F) and **ddi_mapdev_nointercept**(9F) to inform the system of whether or not they are interested in being notified when the user process accesses the mapping. By default, user accesses to newly created mappings will generate a call to the **mapdev_access()** entry point. The driver is always notified of duplications and unmaps.

The device driver can use these interfaces to implement a device context and control user accesses to the device space. Only mappings of type **MAP_PRIVATE** should be used with **ddi_mapdev().**

RETURN VALUES

ddi_mapdev() returns zero on success and non-zero on failure. The return value from **ddi_mapdev()** should be used as the return value for the drivers **segmap()** entry point.

CONTEXT

This routine can be called from user or kernel context only.

SEE ALSO

mmap(2), **munmap**(2), **fork**(2), **segmap**(9E), **mapdev_access**(9E), **mapdev_dup**(9E), **mapdev_free**(9E), **ddi_mapdev_intercept**(9F), **ddi_mapdev_nointercept**(9F), **ddi_mapdev_ctl**(9S),

Writing Device Drivers

NAME

ddi_mapdev_intercept, ddi_mapdev_nointercept – control driver notification of user accesses

SYNOPSIS

#include <sys/sunddi.h>

int ddi_mapdev_intercept(ddi_mapdev_handle_t *handle*, **off_t** *offset*, **off_t** *len***);**

int ddi_mapdev_nointercept(ddi_mapdev_handle_t *handle*, **off_t** *offset*, **off_t** *len***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

handle An opaque pointer to a device mapping handle.

offset An offset in bytes within device memory.

len Length in bytes.

DESCRIPTION

ddi_mapdev_intercept( ) and **ddi_mapdev_nointercept( )** control whether or not user accesses to device mappings created by **ddi_mapdev**(9F) in the specified range will generate calls to the **mapdev_access**(9E) entry point. **ddi_mapdev_intercept( )** tells the system to intercept the user access and notify the driver to invalidate the mapping translations. **ddi_mapdev_nointercept( )** tells the system to not intercept the user access and allow it to proceed by validating the mapping translations.

For both routines, the range to be affected is defined by the *offset* and *len* arguments. Requests affect the entire page containing the *offset* and all pages up to and including the page containing the last byte as indicated by *offset* + *len*.

Supplying a value of **0** for the *len* argument affects all addresses from the *offset* to the end of the mapping. Supplying a value of **0** for the *offset* argument and a value of **0** for *len* argument affect all addresses in the mapping.

To manage a device context, a device driver would call **ddi_mapdev_intercept( )** on the context about to be switched out, switch contexts, and then call **ddi_mapdev_nointercept( )** on the context switched in.

RETURN VALUES

ddi_mapdev_intercept( ) and **ddi_mapdev_nointercept( )** return zero on success and non-zero on failure.

EXAMPLE

The following shows an example of managing a device context that is one page in length.

```
ddi_mapdev_handle_t cur_hdl;

static int
xxmapdev_access(ddi_mapdev_handle_t handle, void *devprivate,
   off_t offset)
{
        int err;

        /* enable access callbacks for the current mapping */
        if (cur_hdl != NULL) {
                if ((err = ddi_mapdev_intercept(cur_hdl, offset, 0)) != 0)
```

```
                return (err);
        }

        /* Switch device context - device dependent*/
        ...

        /* Make handle the new current mapping */
        cur_hdl = handle;

        /*
         * Disable callbacks and complete the access for the
         * mapping that generated this callback.
         */

        return (ddi_mapdev_nointercept(handle, offset, 0));
}
```

CONTEXT

These routines can be called from user or kernel context only.

SEE ALSO

mapdev_access(9E), **ddi_mapdev**(9F)

Writing Device Drivers

NAME

ddi_mem_alloc, ddi_mem_free – allocate and free sequentially accessed memory

SYNOPSIS

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

int ddi_mem_alloc(dev_info_t ∗*dip***, ddi_dma_lim_t** ∗*limits***, u_int** *length***, u_int** *flags***,**
caddr_t ∗*kaddrp***, u_int** ∗*real_length***);**

void ddi_mem_free(caddr_t *kaddr***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ddi_mem_alloc()

dip	A pointer to the device's **dev_info** structure.
limits	A pointer to a DMA limits structure for this device (see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). If this pointer is **NULL,** a default set of DMA limits is assumed.
length	The length in bytes of the desired allocation.
flags	The possible flags **1** and **0** are taken to mean, respectively, wait until memory is available, or do not wait.
kaddrp	On a successful return, ∗*kaddrp* points to the allocated memory.
real_length	The length in bytes that was allocated. Alignment and padding requirements may cause **ddi_mem_alloc()** to allocate more memory than requested in *length*.

ddi_mem_free()

kaddr	The memory returned from a successful call to **ddi_mem_alloc()**.

DESCRIPTION

ddi_mem_alloc() allocates memory for DMA transfers and should be used if the device is performing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as *steaming* access. The allocation will obey the alignment and padding constraints as specified by the *limits* argument and other limits imposed by the system.

Note that you must still use DMA resource allocation functions (see **ddi_dma_setup**(9F)) to establish DMA resources for the memory allocated using **ddi_mem_alloc()**. **ddi_mem_alloc()** returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size. **ddi_dma_setup**(9F) requires the actual length.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using **ddi_dma_sync**(9F) or **ddi_dma_free**(9F) are required.

ddi_mem_free() frees up memory allocated by **ddi_mem_alloc()**.

RETURN VALUES

ddi_mem_alloc() returns:

DDI_SUCCESS Memory successfully allocated.

DDI_FAILURE Allocation failed.

CONTEXT

ddi_mem_alloc() can be called from user or interrupt context, except when *flags* is set to **1**, in which case it can be called from user context only.

SEE ALSO

ddi_dma_free(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **ddi_iopb_alloc**(9F), **ddi_dma_lim_sparc**(9S), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)

Writing Device Drivers

NAME

ddi_peek, ddi_peekc, ddi_peeks, ddi_peekl, ddi_peekd – read a value from a location

SYNOPSIS

```
#include &lt;sys/ddi.h&gt;
#include &lt;sys/sunddi.h&gt;

int ddi_peekc(dev_info_t *dip, char *addr, char *valuep);
int ddi_peeks(dev_info_t *dip, short *addr, short *valuep);
int ddi_peekl(dev_info_t *dip, long *addr, long *valuep);
int ddi_peekd(dev_info_t *dip, longlong_t *addr, longlong_t *valuep);
```

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

addr Virtual address of the location to be examined.

valuep Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the prescence of a device on the system in a drivers **probe**(9E) or **attach**(9E) routines.

RETURN VALUES

DDI_SUCCESS The value at the given virtual address was successfully read, and if *valuep* is non-null, **valuep* will have been updated.

DDI_FAILURE An error occurred whilst trying to read the location, **valuep* is unchanged.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLES

Check to see that the status register of a device is mapped into the kernel address space:

```
if (ddi_peekc(dip, csr, (char *)0) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "Status register not mapped");
        return (DDI_FAILURE);
}
```

Read and log the device type of a particular device:

```
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        ...
        /* map device registers */
        ...

        if (ddi_peekl(dip, id_addr, &id_value) != DDI_SUCCESS) {
                cmn_err(CE_WARN, "%s%d: cannot read device identifier",
                    ddi_get_name(dip), ddi_get_instance(dip));
                goto failure;
        } else
                cmn_err(CE_CONT, "!%s%d: device type 0x%x\n",
                    ddi_get_name(dip), ddi_get_instance(dip), id_value);
        ...
        ...

        ddi_report_dev(dip);
        return (DDI_SUCCESS);

failure:
        /* free any resources allocated */
        ...
        return (DDI_FAILURE);
}
```

SEE ALSO

attach(9E), **probe**(9E), **ddi_poke**(9F)

Writing Device Drivers

NAME

ddi_poke, ddi_pokec, ddi_pokes, ddi_pokel, ddi_poked – write a value to a location

SYNOPSIS

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

int ddi_pokec(dev_info_t **dip*, **char** **addr*, **char** *value*);
int ddi_pokes(dev_info_t **dip*, **short** **addr*, **short** *value*);
int ddi_pokel(dev_info_t **dip*, **long** **addr*, **long** *value*);
int ddi_poked(dev_info_t **dip*, **longlong_t** **addr*, **longlong_t** *value*);

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

addr Virtual address of the location to be written to.

value Value to be written to the location.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's **probe**(9E) or **attach**(9E) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the **ddi_peek**(9F) routines instead if possible.

RETURN VALUES

DDI_SUCCESS The value was successfully written to the given virtual address.

DDI_FAILURE An error occurred while trying to write to the location.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

attach(9E), **probe**(9E), **ddi_peek**(9F)

Writing Device Drivers

NAME

ddi_prop_create, ddi_prop_modify, ddi_prop_remove, ddi_prop_remove_all, ddi_prop_undefine – create, remove, or modify properties for leaf device drivers

SYNOPSIS

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

int ddi_prop_create(dev_t *dev*, **dev_info_t** **dip*, **int** *flags*, **char** **name*, **caddr_t** *valuep*, **int** *length***);**

int ddi_prop_undefine(dev_t *dev*, **dev_info_t** **dip*, **int** *flags*, **char** **name***);**

int ddi_prop_modify(dev_t *dev*, **dev_info_t** **dip*, **int** *flags*, **char** **name*, **caddr_t** *valuep*, **int** *length***);**

int ddi_prop_remove(dev_t *dev*, **dev_info_t** **dip*, **char** **name***);**

void ddi_prop_remove_all(dev_info_t **dip***);**

ARGUMENTS

ddi_prop_create( )

dev	**dev_t** of the device.
dip	**dev_info_t** pointer of the device.
flags	flag modifiers. The only possible flag value is **DDI_PROP_CANSLEEP**: Memory allocation may sleep.
name	name of property.
valuep	pointer to property value.
length	property length.

ddi_prop_undefine( )

dev	**dev_t** of the device.
dip	**dev_info_t** pointer of the device.
flags	flag modifiers. The only possible flag value is **DDI_PROP_CANSLEEP**: Memory allocation may sleep.
name	name of property.

ddi_prop_modify( )

dev	**dev_t** of the device.
dip	**dev_info_t** pointer of the device.
flags	flag modifiers. The only possible flag value is **DDI_PROP_CANSLEEP**: Memory allocation may sleep.
name	name of property.
valuep	pointer to property value.
length	property length.

ddi_prop_remove( )

dev	**dev_t** of the device.
dip	**dev_info_t** pointer of the device.
name	name of property.

ddi_prop_remove_all( )

dip	**dev_info_t** pointer of the device.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses **ddi_getproplen**(9F) to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver's property list associated with *dip.*

Property definitions are stacked; they are added to the beginning of the driver's property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be return to the caller.

ddi_prop_create( )

ddi_prop_create( ) adds a property to the device's property list. If the property is not associated with any particular *dev* but is associated with the physical device itself, then the argument *dev* should be the special device **DDI_DEV_T_NONE.** If you do not have a *dev* for your device (for example during **attach**(9E) time), you can create one using **makedevice**(9F) with a major number of DDI_MAJOR_T_UNKNOWN. **ddi_prop_create**( ) will then make the correct *dev* for your device.

For boolean properties, you must set *length* to **0.** For all other properties, the *length* argument must be set to the number of bytes used by the data structure representing the property being created.

Note that creating a property involves allocating memory for the property list, the property name and the property value. If *flags* does not contain **DDI_PROP_CANSLEEP, ddi_prop_create**( ) returns **DDI_PROP_NO_MEMORY** on memory allocation failure or **DDI_SUCCESS** if the allocation succeeded. If **DDI_PROP_CANSLEEP** was set, the caller may sleep until memory becomes available.

ddi_prop_undefine( )

ddi_prop_undefine( ) is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. See **ddi_prop_op**(9F).

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as **ddi_prop_create**( ).

ddi_prop_modify( )

ddi_prop_modify( ) modifies the length and the value of a property. If **ddi_prop_modify**( ) finds the property in the driver's property list, allocates memory for the property value and returns **DDI_PROP_SUCCESS.** If the property was not found, the function returns **DDI_PROP_NOT_FOUND.**

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as **ddi_prop_create**( ).

ddi_prop_remove()

ddi_prop_remove( ) unlinks a property from the device's property list. If **ddi_prop_remove**( ) finds the property (an exact match of both *name*and *dev*), it unlinks the property, frees its memory, and returns **DDI_PROP_SUCCESS,** otherwise, it returns **DDI_PROP_NOT_FOUND.**

ddi_prop_remove_all()

ddi_prop_remove_all( ) removes the properties of all the **dev_t**'s associated with the *dip*. It is called before unloading a driver.

RETURN VALUES

ddi_prop_create ( )

DDI_PROP_SUCCESS	on success.
DDI_PROP_NO_MEMORY	on memory allocation failure.
DDI_PROP_INVAL_ARG	if an attempt is made to create a property with *dev* equal to **DDI_DEV_T_ANY** or if *name* is NULL or *name* is the NULL string.

ddi_prop_undefine( )

DDI_PROP_SUCCESS	on success.
DDI_PROP_NO_MEMORY	on memory allocation failure.
DDI_PROP_INVAL_ARG	if an attempt is made to create a property with *dev* **DDI_DEV_T_ANY** or if *name* is NULL or *name* is the NULL string.

ddi_prop_modify( )

DDI_PROP_SUCCESS	on success.
DDI_PROP_NO_MEMORY	on memory allocation failure.
DDI_PROP_INVAL_ARG	if an attempt is made to create a property with *dev* equal to **DDI_DEV_T_ANY** or if *name* is NULL or *name* is the NULL string.
DDI_PROP_NOT_FOUND	on property search failure.

ddi_prop_remove( )

DDI_PROP_SUCCESS	on success.
DDI_PROP_INVAL_ARG	if an attempt is made to create a property with *dev* equal to **DDI_DEV_T_ANY** or if *name* is NULL or *name* is the NULL string.
DDI_PROP_NOT_FOUND	on property search failure.

CONTEXT

If **DDI_PROP_CANSLEEP** is set, these functions can only be called from user context; otherwise, they can be called from interrupt or user context.

EXAMPLES

Create a property called *nblocks* for each partition on a disk.

```
for (minor = 0; minor < 8; minor ++) {
        (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),
          dev, DDI_PROP_CANSLEEP, "nblocks", 8192, sizeof (int));
        ...
}
```

SEE ALSO

attach(9E), **ddi_prop_op**(9F), **makedevice**(9F), **driver.conf**(4)

Writing Device Drivers

NAME

ddi_prop_op, ddi_getprop, ddi_getlongprop, ddi_getlongprop_buf, ddi_getproplen – get property information for leaf device drivers

SYNOPSIS

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

int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp);

int ddi_getprop(dev_t dev, dev_info_t *dip, int flags,
    char *name, int defvalue);

int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags,
    char *name, caddr_t valuep, int *lengthp);

int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags,
    char *name, caddr_t valuep, int *lengthp);

int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags,
    char *name, int *lengthp);
```

ARGUMENTS

dev Device number associated with property or **DDI_DEV_T_ANY** as the *wildcard* device number.

dip Pointer to a device info node.

prop_op Property operator.

flags Possible flag values are some combination of:

DDI_PROP_DONTPASS
do not pass request to parent device information node if property not found

DDI_PROP_CANSLEEP
the routine may sleep while allocating memory

DDI_PROP_NOTPROM
do not look at PROM properties (ignored on architectures that do not support PROM properties).

name String containing the name of the property.

valuep If *prop_op* is **PROP_LEN_AND_VAL_BUF,** this should be a pointer to the users buffer. If *prop_op* is **PROP_LEN_AND_VAL_ALLOC,** this should be the *address* of a pointer.

lengthp On exit, **lengthp* will contain the property length. If *prop_op* is **PROP_LEN_AND_VAL_BUF** then before calling **ddi_prop_op**( ), *lengthp* should point to an **int** that contains the length of callers buffer.

defvalue The value that **ddi_getprop**( ) returns if the property is not found.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_prop_op( ) gets arbitrary-size properties for leaf devices. The routine searches the device's property list. If it does not find the property at the device level, it examines the *flags* argument, and if **DDI_PROP_DONTPASS** is set, then **ddi_prop_op**( ) returns **DDI_PROP_NOT_FOUND.** Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns **DDI_PROP_UNDEFINED.** Otherwise it returns either the property length, or both the length and value of the property to the caller via the *valuep* and *lengthp* pointers, depending on the value of *prop_op*, as described below, and returns **DDI_PROP_SUCCESS.** If a property cannot be found at all, **DDI_PROP_NOT_FOUND** is returned.

Usually, the *dev* argument should be set to the actual device number that this property applies to. However, if the *dev* argument is **DDI_DEV_T_ANY,** the *wildcard dev*, then **ddi_prop_op**( ) will match the request based on *name* only (regardless of the actual *dev* the property was created with). This property / dev match is done according to the property search order which is to first search software properties created by the driver in *last-in, first-out* (LIFO) order, next search software properties created by the *system* in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the *prop_op* argument. If *prop_op* is **PROP_LEN,** then **ddi_prop_op**( ) just sets the callers length, ∗*lengthp*, to the property length and returns the value **DDI_PROP_SUCCESS** to the caller. The *valuep* argument is not used in this case. Property lengths are **0** for boolean properties, **sizeof (int)** for integer properties, and size in bytes for long (variable size) properties.

If *prop_op* is **PROP_LEN_AND_VAL_BUF,** then *valuep* should be a pointer to a user-supplied buffer whose length should be given in ∗*lengthp* by the caller. If the requested property exists, **ddi_prop_op**( ) first sets ∗*lengthp* to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns **DDI_PROP_SUCCESS.** If the named property exists but the buffer supplied is too small to hold it, it returns **DDI_PROP_BUF_TOO_SMALL.**

If *prop_op* is **PROP_LEN_AND_VAL_ALLOC,** and the property is found, **ddi_prop_op**( ) sets ∗*lengthp* to the property length. It then attempts to allocate a buffer to return to the caller using the **kmem_alloc**(9F) routine, so that memory can be later recycled using **kmem_free**(9F). The driver is expected to call **kmem_free**( ) with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets ∗*valuep* to point to the allocated buffer, copies the property value into the buffer and returns **DDI_PROP_SUCCESS.** Otherwise, it returns **DDI_PROP_NO_MEMORY.** Note that the *flags* argument may affect the behavior of memory allocation in **ddi_prop_op**( ). In particular, if **DDI_PROP_CANSLEEP** is set, then the routine will wait until memory is available to copy the requested property.

ddi_getprop( ) returns boolean and integer-size properties. It is a convenience wrapper for **ddi_prop_op**( ) with *prop_op* set to **PROP_LEN_AND_VAL_BUF,** and the buffer is provided by the wrapper. By convention, this function returns a **1** for boolean (zero-length) properties.

ddi_getlongprop( ) returns arbitrary-size properties. It is a convenience wrapper for **ddi_prop_op**( ) with *prop_op* set to **PROP_LEN_AND_VAL_ALLOC,** so that the routine will allocate space to hold the buffer that will be returned to the caller via ∗*valuep*.

ddi_getlongprop_buf( ) returns arbitrary-size properties. It is a convenience wrapper for **ddi_prop_op**( ) with *prop_op* set to **PROP_LEN_AND_VAL_BUF** so the user must supply a buffer.

ddi_getproplen( ) returns the length of a given property. It is a convenience wrapper for **ddi_prop_op**( ) with *prop_op* set to **PROP_LEN.**

RETURN VALUES

ddi_prop_op( )
ddi_getlongprop( )
ddi_getlongprop_buf( )
ddi_getproplen( ) return:

DDI_PROP_SUCCESS	Property found and returned.
DDI_PROP_NOT_FOUND	Property not found.
DDI_PROP_UNDEFINED	Property already explicitly undefined.
DDI_PROP_NO_MEMORY	Property found, but unable to allocate memory. *lengthp* points to the correct property length.
DDI_PROP_BUF_TOO_SMALL	Property found, but the supplied buffer is too small. *lengthp* points to the correct property length.

ddi_getprop( ) returns:

The value of the property or the value passed into the routine as **defvalue** if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

CONTEXT

These functions can be called from user or interrupt context, provided **DDI_PROP_CANSLEEP** is not set; if it is set, they can be called from user context only.

SEE ALSO

ddi_prop_create(9F), **kmem_alloc**(9F), **kmem_free**(9F)

Writing Device Drivers

NAME

ddi_remove_minor_node – remove a minor node for this dev_info

SYNOPSIS

void ddi_remove_minor_node(dev_info_t **dip*, **char** **name***)**

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

name The name of this minor device. If *name* is NULL then remove all minor data structures from this dev_info.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_remove_minor_node( ) removes a data structure from the linked list of minor data structures that is pointed to by the dev_info structure for this driver.

EXAMPLES

This will remove a data structure describing a minor device called **foo** which is linked into the dev_info structure pointed to by **dip**.

ddi_remove_minor_node(dip, "foo");

SEE ALSO

attach(9E), **detach**(9E), **ddi_create_minor_node**(9F)

Writing Device Drivers

NAME

ddi_report_dev – announce a device

SYNOPSIS

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

void ddi_report_dev(dev_info_t *dip);
```

ARGUMENTS

dip a pointer the device's **dev_info** structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_report_dev( ) prints a banner at boot time, announcing the device pointed to by *dip*. The banner is always placed in the system logfile (displayed by **dmesg**(1M)), but is only displayed on the console if the system was booted with the verbose **(–v)** argument.

CONTEXT

ddi_report_dev( ) can be called from user or interrupt context.

SEE ALSO

dmesg(1M), **kernel**(1M)

Writing Device Drivers

NAME

ddi_root_node – get the root of the dev_info tree

SYNOPSIS

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

dev_info_t *ddi_root_node(void)

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_root_node( ) returns a pointer to the root node of the device information tree.

RETURN VALUES

ddi_root_node( ) returns a pointer to a device information structure.

CONTEXT

ddi_root_node( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

ddi_segmap – map a segment

SYNOPSIS

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

int ddi_segmap(dev_t *dev***, off_t** *offset***, struct as** **asp,* **caddr_t** **addrp,* **off_t** *len,* **u_int** *prot***, u_int** *maxprot,* **u_int** *flags,* **cred_t** **credp***);**

ARGUMENTS

dev	Device number.
offset	Offset into device.
asp	Pointer to **as** (address space) structure.
addrp	Pointer to virtual address.
len	Length in bytes.
prot	Protection.
maxprot	Protection.
flags.	Flags.
credp	Pointer to user credential structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

ddi_segmap( ) provides the default segment driver. It calls the driver's **mmap**(9E) routine to validate the range to be mapped.

It is typically used as the **segmap**(9E) entry in the **cb_ops** structure for those devices that do not need to provide their own segment driver, and is not usually called directly by drivers. However, some drivers may have their own **segmap**(9E) entry to do some initial processing on the parameters (such as picking a virtual address, if the user did not provide one), and then call **ddi_segmap**( ) to establish the default memory segment.

RETURN VALUES

ddi_segmap( ) returns:

0	on success.
non-zero	on failure. In particular, it returns ENXIO if the range to be mapped is invalid.

CONTEXT

ddi_segmap( ) can be called from user or interrupt context.

SEE ALSO

mmap(9E), **segmap**(9E)

Writing Device Drivers

NAME

ddi_slaveonly – tell if a device is installed in a slave access only location

SYNOPSIS

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

int ddi_slaveonly(dev_info_t **dip***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dip A pointer to the device's **dev_info** structure.

DESCRIPTION

ddi_slaveonly tells the caller if the bus, or part of the bus that the device is installed on, does not permit the device to become a DMA master, that is, whether the device has been installed in a slave access only slot.

RETURN VALUES

DDI_SUCCESS
The device has been installed in a slave access only location.

DDI_FAILURE
The device has *not* been installed in a slave access only location.

CONTEXT

ddi_slaveonly can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

ddi_soft_state, ddi_get_soft_state, ddi_soft_state_fini, ddi_soft_state_free, ddi_soft_state_init, ddi_soft_state_zalloc – driver soft state utility routines

SYNOPSIS

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

void *ddi_get_soft_state(void **state*, **int** *item***);**

void ddi_soft_state_fini(void ***state_p***);**

void ddi_soft_state_free(void **state*, **int** *item***);**

int ddi_soft_state_init(void ***state_p*, **size_t** *size*,
    size_t *n_items***);**

int ddi_soft_state_zalloc(void **state*, **int** *item***);**

ARGUMENTS

state_p — Address of the opaque state pointer which will be initialized by **ddi_soft_state_init**() to point to implementation dependent data.

size — Size of the item which will be allocated by subsequent calls to **ddi_soft_state_zalloc**().

n_items — A hint of the number of items which will be preallocated; zero is allowed.

state — An opaque pointer to implementation-dependent data that describes the soft state.

item — The item number for the state structure; usually the instance number of the associated devinfo node.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

Most device drivers maintain state information with each instance of the device they control; for example, a soft copy of a device control register, a mutex that must be held while accessing a piece of hardware, a partition table, or a unit structure. These utility routines are intended to help device drivers manage the space used by the driver to hold such state information.

For example, if the driver holds the state of each instance in a single state structure, these routines can be used to dynamically allocate and deallocate a separate structure for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, *state_p*, which the implementation uses as a place to hang a set of per-driver structures; everything else is managed by these routines.

The routine **ddi_soft_state_init**() is usually called in the drivers **_init**(9E) routine to initialize the state pointer, set the size of the soft state structure, and to allow the driver to pre-allocate a given number of such structures if required.

The routine **ddi_soft_state_zalloc**() is usually called in the drivers **attach**(9E) routine. The routine is passed an item number which is used to refer to the structure in subsequent calls to **ddi_get_soft_state**() and **ddi_soft_state_free**(). The item number is usually just the instance number of the devinfo node, obtained with **ddi_get_instance**(9F). The routine attempts to allocate space for the new structure, and if the space allocation was successful, **DDI_SUCCESS** is returned to the caller.

A pointer to the space previously allocated for a soft state structure can be obtained by calling **ddi_get_soft_state**() with the appropriate item number.

The space used by a given soft state structure can be returned to the system using **ddi_soft_state_free**(). This routine is usually called from the drivers **detach**(9E) entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using **ddi_soft_state_fini**(). This routine can be called from the drivers **_fini**(9E) routine.

The **ddi_soft_state_zalloc**(), **ddi_soft_state_free**() and **ddi_get_soft_state**() routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

RETURN VALUES

ddi_get_soft_state():

NULL The requested state structure was not allocated at the time of the call.

pointer The pointer to the state structure.

ddi_soft_state_init():

0 The allocation was successful.

EINVAL Either the *size* parameter was zero, or the *state_p* parameter was invalid.

ddi_soft_state_zalloc():

DDI_SUCCESS The allocation was successful.

DDI_FAILURE The routine failed to allocate the storage required; either the *state* parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

CONTEXT

ddi_soft_state_init(), and **ddi_soft_state_alloc**() can be called from user context only, since they may internally call **kmem_zalloc**(9F) with the **KM_SLEEP** flag.

The **ddi_soft_state_fini**(), **ddi_soft_state_free**() and **ddi_get_soft_state**() routines can be called from any driver context.

EXAMPLE

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the drivers data structures.

```
typedef struct {
        volatile caddr_t *csr;          /* device registers */
        kmutex_t      csr_mutex;        /* protects 'csr' field */
```

```
        unsigned int    state;
        dev_info_t      *dip;           /* back pointer to devinfo */
} devstate_t;

static void *statep;

int
_init(void)
{
        int error;

        error = ddi_soft_state_init(&statep, sizeof (devstate_t), 0);
        if (error != 0)
                return (error);
        if ((error = mod_install(&modlinkage)) != 0)
                ddi_soft_state_fini(&statep);
        return (error);
}

int
_fini(void)
{
        int error;

        if ((error = mod_remove(&modlinkage)) != 0)
                return (error);
        ddi_soft_state_fini(&statep);
        return (0);
}

static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int instance;
        devstate_t *softc;

        switch (cmd) {
        case DDI_ATTACH:
                instance = ddi_get_instance(dip);
                if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
                        return (DDI_FAILURE);
                softc = ddi_get_soft_state(statep, instance);
                softc->dip = dip;
                ...
                return (DDI_SUCCESS);
```

```
        default:
                return (DDI_FAILURE);
        }
}

static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        int instance;

        switch (cmd) {

        case DDI_DETACH:
                instance = ddi_get_instance(dip);
                ...
                ddi_soft_state_free(statep, instance);
                return (DDI_SUCCESS);

        default:
                return (DDI_FAILURE);
        }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
        devstate_t *softc;
        int        instance;

        instance = getminor(*devp);
        if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
                return (ENXIO);
        ...
        softc->state |= XX_IN_USE;
        ...
        return (0);
}
```

SEE ALSO

_fini(9E), **_init**(9E), **attach**(9E), **detach**(9E), **ddi_get_instance**(9F), **getminor**(9F), **kmem_zalloc**(9F)

Writing Device Drivers

WARNINGS

There is no attempt to validate the **item** parameter given to **ddi_soft_state_zalloc**(); other than it must be a positive signed integer. Therefore very large item numbers may cause the driver to hang forever waiting for virtual memory resources that can never be satisfied.

NOTES

If necessary, a hierarchy of state structures can be constructed by embedding state pointers in higher order state structures.

DIAGNOSTICS

All of the messages described below usually indicate bugs in the driver and should not appear in normal operation of the system.

WARNING: ddi_soft_state_zalloc: bad handle
WARNING: ddi_soft_state_free: bad handle
WARNING: ddi_soft_state_fini: bad handle

The implementation-dependent information kept in the state variable is corrupt.

WARNING: ddi_soft_state_free: null handle
WARNING: ddi_soft_state_fini: null handle

The routine has been passed a null or corrupt state pointer. Check that **ddi_soft_state_init**() has been called.

WARNING: ddi_soft_state_free: item %d not in range [0..%d]

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.

NAME

delay – delay execution for a specified number of clock ticks

SYNOPSIS

#include <sys/ddi.h>

void delay(long *ticks***);**

ARGUMENTS

ticks The number of clock cycles to delay.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

delay( ) provides a mechanism for a driver to delay its execution for a given period of time. Since the speed of the clock varies among systems, drivers should base their time values on microseconds and use **drv_usectohz**(9F) to convert microseconds into clock ticks.

delay( ) uses **timeout**(9F) to schedule an internal function to be called after the specified amount of time has elapsed. **delay**( ) then waits until the function is called.

delay( ) does not busy-wait. If busy-waiting is required, use **drv_usecwait**(9F).

CONTEXT

delay( ) can be called from user context only.

EXAMPLE

Before a driver I/O routine allocates buffers and stores any user data in them, it checks the status of the device (line 12). If the device needs manual intervention (such as, needing to be refilled with paper), a message is displayed on the system console (line 14). The driver waits an allotted time (line 17) before repeating the procedure.

```
 1 struct  device {                          /* layout of physical device registers */
 2       int     control;                    /* physical device control word */
 3       int     status;                     /* physical device status word */
 4       short   xmit_char;                  /* transmit character to device */
 5 };
 6
 7
   . . .
 9                                           /* get device registers  */
10     register struct device *rp = ...
11
12     while (rp->status & NOPAPER) { /* while printer is out of paper */
13                                             /* display message and ring bell */
                                               /* on system console */
14        cmn_err(CE_WARN, "^xx_write: NO PAPER in printer %d\007",
15                    (getminor(dev) & 0xf));
16        /* wait one minute and try again */
17        delay(60 * drv_usectohz(1000000));
18     }
```

SEE ALSO

biodone(9F), **biowait**(9F), **drv_hztousec**(9F), **drv_usectohz**(9F), **drv_usecwait**(9F), **timeout**(9F), **untimeout**(9F)

Writing Device Drivers

NAME disksort – single direction elevator seek sort for buffers

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

void disksort(struct diskhd *∗dp*, **struct buf** *∗bp***)**

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION The function **disksort**( ) sorts a pointer to a buffer into a single forward linked list headed by the **av_forw** element of the argument ∗**dp**.

It uses a one-way elevator algorithm that sorts buffers into the queue in ascending order based upon a key value held in the argument buffer structure element **b_resid**.

This value can either be the driver calculated cylinder number for the I/O request described by the buffer argument, or simply the absolute logical block for the I/O request, depending on how fine grained the sort is desired to be or how applicable either quantity is to the device in question.

The head of the linked list is found by use of the **av_forw** structure element of the argument ∗**dp**. The tail of the linked list is found by use of the **av_back** structure element of the argument ∗**dp**. The **av_forw** element of the ∗**bp** argument is used by **disksort**( ) to maintain the forward linkage. The value at the head of the list presumably indicates the currently active disk area.

ARGUMENTS

dp A pointer to a **diskhd** structure. A **diskhd** structure is essentially identical to head of a buffer structure (see **buf**(9S)). The only defined items of interest for this structure are the **av_forw** and **av_back** structure elements which are used to maintain the front and tail pointers of the forward linked I/O request queue.

bp A pointer to a buffer structure. Typically this is the I/O request that the driver receives in its strategy routine (see **strategy**(9E)). The driver is responsible for initializes the **b_resid** structure element to a meaningful sort key value prior to calling **disksort**( ).

WARNING **disksort**( ) does no locking. Therefore, any locking is completely the responsibility of the caller.

CONTEXT This function can be called from user or interrupt context.

SEE ALSO **strategy**(9E), **buf**(9S)

Writing Device Drivers

NAME

drv_getparm – retrieve kernel state information

SYNOPSIS

#include <sys/ddi.h>

int drv_getparm(unsigned long *parm***, unsigned long** ∗*value_p***);**

ARGUMENTS

parm The kernel parameter to be obtained. Possible values are:

LBOLT Read the value of **lbolt**. (**lbolt** is an integer that represents the number of clock ticks since the last system reboot. This value is used as a counter or timer inside the system kernel.)

PPGRP Read the process group identification number. This number determines which processes should receive a **HANGUP** or **BREAK** signal when detected by a driver.

UPROCP Read the process table token value.

PPID Read process identification number.

PSID Read process session identification number.

TIME Read time in seconds.

UCRED Return a pointer to the caller's credential structure.

value_p A pointer to the data space in which the value of the parameter is to be copied.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

drv_getparm( ) function verifies that *parm* corresponds to a kernel parameter that may be read. If the value of *parm* does not correspond to a parameter or corresponds to a parameter that may not be read, **-1** is returned. Otherwise, the value of the parameter is stored in the data space pointed to by *value_p*.

drv_getparm( ) does not explicitly check to see whether the device has the appropriate context when the function is called and the function does not check for correct alignment in the data space pointed to by *value_p*. It is the responsibility of the driver writer to use this function only when it is appropriate to do so and to correctly declare the data space needed by the driver.

RETURN VALUES

drv_getparm( ) returns **0** to indicate success, **–1** to indicate failure. The value stored in the space pointed to by *value_p* is the value of the parameter if **0** is returned, or undefined if –**1** is returned. –**1** is returned if you specify a value other than **LBOLT**, **PPGRP**, **PPID**, **PSID**, **TIME**, **UCRED**, or **UPROCP**. Always check the return code when using this function.

CONTEXT

drv_getparm( ) can be called from user context only when using **PPGRP**, **PPID**, **PSID**, **UCRED**, or **UPROCP**. It can be called from user or interrupt context when using the **LBOLT** or **TIME** argument.

SEE ALSO

buf(9S)

Writing Device Drivers

NAME

drv_hztousec – convert clock ticks to microseconds

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

clock_t drv_hztousec(clock_t *hertz* **);**

ARGUMENTS

hertz The number of clock ticks to convert.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

drv_hztousec( ) converts into microseconds the time expressed by *hertz*, which is in system clock ticks.

The kernel variable **lbolt**, which is (only) readable through **drv_getparm**(9F), is the length of time the system has been up since boot and is expressed in clock ticks. Drivers often use the value of **lbolt** before and after an I/O request to measure the amount of time it took the device to process the request. **drv_hztousec( )** can be used by the driver to convert the reading from clock ticks to a known unit of time.

RETURN VALUES

The number of microseconds equivalent to the *hertz* argument.
No error value is returned. If the microsecond equivalent to *hertz* is too large to be represented as a **clock_t ,** then the maximum **clock_t** value will be returned.

CONTEXT

drv_hztousec( ) can be called from user or interrupt context.

SEE ALSO

drv_getparm(9F), **drv_usectohz**(9F), **drv_usecwait**(9F)

Writing Device Drivers

NAME

drv_priv – determine driver privilege

SYNOPSIS

#include <sys/types.h>
#include <sys/cred.h>
#include <sys/ddi.h>

int drv_priv(cred_t **cr*);

ARGUMENTS

cr Pointer to the user credential structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

drv_priv( ) provides a general interface to the system privilege policy. It determines whether the credentials supplied by the user credential structure pointed to by *cr* identify a privileged process. This function should only be used when file access modes and special minor device numbers are insufficient to provide protection for the requested driver function. It is intended to replace all calls to **suser**( ) and any explicit checks for effective **user ID = 0** in driver code.

RETURN VALUES

This routine returns **0** if it succeeds, **EPERM** if it fails.

CONTEXT

drv_priv( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

drv_usectohz – convert microseconds to clock ticks

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

clock_t drv_usectohz(clock_t *microsecs***);**

ARGUMENTS

microsecs The number of microseconds to convert.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

drv_usectohz( ) converts a length of time expressed in microseconds to a number of system clock ticks. The time arguments to **timeout**(9F) and **delay**(9F) are expressed in clock ticks.

drv_usectohz( ) is a portable interface for drivers to make calls to **timeout**(9F) and **delay**(9F) and remain binary compatible should the driver object file be used on a system with a different clock speed (a different number of ticks in a second).

RETURN VALUES

The value returned is the number of system clock ticks equivalent to the *microsecs* argument. No error value is returned. If the clock tick equivalent to *microsecs* is too large to be represented as a **clock_t**, then the maximum **clock_t** value will be returned.

CONTEXT

drv_usectohz( ) can be called from user or interrupt context.

SEE ALSO

delay(9F), **drv_hztousec**(9F), **timeout**(9F)

Writing Device Drivers

NAME

drv_usecwait – busy-wait for specified interval

SYNOPSIS

#include <sys/types.h>
#include <sys/ddi.h>

void drv_usecwait(clock_t *microsecs***);**

ARGUMENTS

microsecs The number of microseconds to busy-wait.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

drv_usecwait( ) gives drivers a means of busy-waiting for a specified microsecond count. The amount of time spent busy-waiting may be greater than the microsecond count but will minimally be the number of microseconds specified.

delay(9F) can be used by a driver to delay for a specified number of system ticks, but it has two limitations. First, the granularity of the wait time is limited to one clock tick, which may be more time than is needed for the delay. Second, **delay**(9F) may only be invoked from user context and hence cannot be used at interrupt time or system initialization.

Often, drivers need to delay for only a few microseconds, waiting for a write to a device register to be picked up by the device. In this case, even in user context, **delay**(9F) produces too long a wait period.

CONTEXT

drv_usecwait( ) can be called from user or interrupt context.

SEE ALSO

delay(9F), **timeout**(9F), **untimeout**(9F)

Writing Device Drivers

NOTES

The driver wastes processor time by making this call since **drv_usecwait**( ) does not block but simply busy-waits. The driver should only make calls to **drv_usecwait**( ) as needed, and only for as much time as needed. **drv_usecwait**( ) does not mask out interrupts.

NAME

dupb – duplicate a message block descriptor

SYNOPSIS

#include <sys/stream.h>

mblk_t *dupb(mblk_t **bp*);

ARGUMENTS

bp Pointer to the message block to be duplicated. **mblk_t** is an instance of the **msgb**(9S) structure.

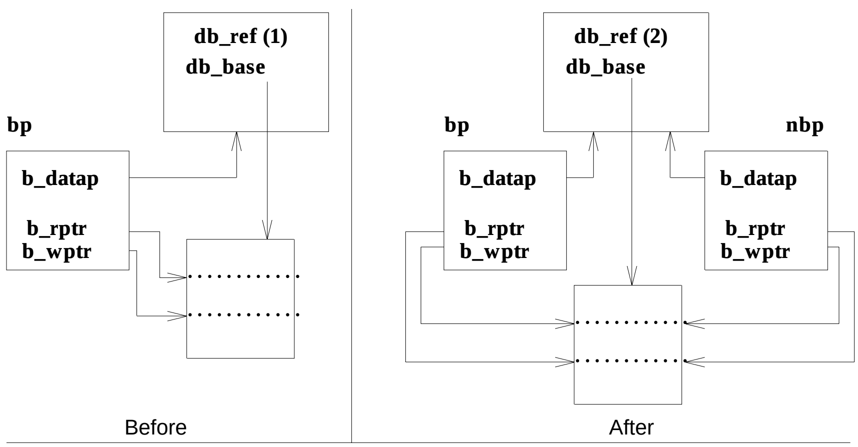
INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

dupb( ) creates a new **mblk_t** structure to reference the message block pointed to by *bp*. Unlike **copyb**(9F), **dupb** does not copy the information in the data block, but creates a new structure to point to it.

The following figure shows how the **db_ref** field of the **dblk_t** structure has been changed from **1** to **2**, reflecting the increase in the number of references to the data block. The new **mblk_t** contains the same information as the first. Note that **b_rptr** and **b_wptr** are copied from *bp*, and that **db_ref** is incremented.



nbp=dupb(bp);

RETURN VALUES

If successful, **dupb** returns a pointer to the new message block. Otherwise, it returns a **NULL** pointer.

CONTEXT

dupb( ) can be called from user or interrupt context.

EXAMPLE

This **srv**(9E) (service) routine adds a header to all **M_DATA** messages before passing them along. The message block for the header was allocated elsewhere. For each message on the queue, if it is a priority message, pass it along immediately (lines 9–10).

Otherwise, if it is anything other than an **M_DATA** message (line 11), and if it can be sent along (line 12), then do so (line 13). Otherwise, put the message back on the queue and return (lines 15–16). For all **M_DATA** messages, first check to see if the stream is flow-controlled (line 19). If it is, put the message back on the queue and return (line 22); if it is not, the header block is duplicated (line 20). If **dupb** fails, the service routine is rescheduled in one tenth of a second with **timeout** and then we return (lines 23–24). If **dupb** succeeds, link the **M_DATA** message to it (line 26) and pass it along (line 27). **dupb** can be used here instead of **copyb**(9F) because the contents of the header block are not changed.

Note that this example ignores issues related to cancelling outstanding timeouts at close time.

```
 1 xxxsrv(q)
 2     queue_t *q;
 3 {
 4         mblk_t *mp;
 5         mblk_t *bp;
 6         extern mblk_t *hdr;
 7
 8         while ((mp = getq(q)) != NULL) {
 9                 if (mp->b_datap->db_type >= QPCTL) {
10                         putnext(q, mp);
11                 } else if (mp->b_datap->db_type != M_DATA) {
12                         if (canputnext(q))
13                                 putnext(q, mp);
14                         else {
15                                 putbq(q, mp);
16                                 return;
17                         }
18                 } else {  /* M_DATA */
19                         if (canputnext(q)) {
20                                 bp = dupb(hdr);
21                                 if (bp == NULL) {
22                                         putbq(q, mp);
23                                         timeout(qenable, (long)q, drv_usectohz(100000));
24                                         return;
25                                 }
26                                 linkb(bp, mp);
27                                 putnext(q, bp);
28                         } else {
29                                 putbq(q, mp);
30                                 return;
31                         }
32                 }
33         }
34 }
```

SEE ALSO

copyb(9F), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

dupmsg – duplicate a message

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**dupmsg(mblk_t** ∗*mp***);**

ARGUMENTS

mp Pointer to the message.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

dupmsg( ) forms a new message by copying the message block descriptors pointed to by *mp* and linking them. **dupb**(9F) is called for each message block. The data blocks themselves are not duplicated.

RETURN VALUES

If successful, **dupmsg( )** returns a pointer to the new message block. Otherwise, it returns a **NULL** pointer.

CONTEXT

dupmsg( ) can be called from user or interrupt context.

EXAMPLE

See **copyb**(9F) for an example using **dupmsg( )**.

SEE ALSO

copyb(9F), **copymsg**(9F), **dupb**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

enableok – reschedule a queue for service

SYNOPSIS

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

void enableok(queue_t **q*);

ARGUMENTS

q A pointer to the queue to be rescheduled.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

enableok( ) enables queue *q* to be rescheduled for service. It reverses the effect of a previous call to **noenable**(9F) on *q* by turning off the **QNOENB** flag in the queue.

CONTEXT

enableok( ) can be called from user or interrupt context.

EXAMPLE

The **qrestart**( ) routine uses two STREAMS functions to restart a queue that has been disabled. The **enableok**( ) function turns off the **QNOENB** flag, allowing the **qenable**(9F) to schedule the queue for immediate processing.

```
void
qrestart(rdwr_q)
        register queue_t *rdwr_q;
{
        enableok(rdwr_q);
        /* re-enable a queue that has been disabled */
        (void) qenable(rdwr_q);
}
```

SEE ALSO

noenable(9F), **qenable**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

esballoc – allocate a message block using a caller-supplied buffer

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**esballoc(unsigned char** ∗*base,* **int** *size,* **int** *pri,* **frtn_t** ∗*fr_rtnp***);**

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

ARGUMENTS

base Address of user supplied data buffer.

size Number of bytes in data buffer.

pri Priority of allocation request (to be used by **allocb**(9F) function, called by **esballoc**( )).

fr_rtnp Free routine data structure.

DESCRIPTION

esballoc( ) creates a STREAMS message and attaches a user-supplied data buffer in place of a STREAMS data buffer. It calls **allocb**(9F) to get a message and data block header only. The user-supplied data buffer, pointed to by *base*, is used as the data buffer for the message.

When **freeb**(9F) is called to free the message, the driver's message freeing routine (referenced through the **free_rtn** structure) is called, with appropriate arguments, to free the data buffer.

The **free_rtn** structure includes the following members:

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

Instead of requiring a specific number of arguments, the **free_arg** field is defined of type **char** ∗. This way, the driver can pass a pointer to a structure if more than one argument is needed.

The method by which **free_func** is called is implementation-specific. The module writer must not assume that free_func will or will not be called directly from STREAMS utility routines like **freeb**(9F) which free a message block.

free_func must not call another modules put procedure nor attempt to acquire a private module lock which may be held by another thread across a call to a STREAMS utility routine which could free a message block. Otherwise, the possibility for lock recursion and / or deadlock exists.

free_func must not access any dynamically allocated data structure that might no longer exist when it runs.

RETURN VALUES

On success, a pointer to the newly allocated message block is returned. On failure, **NULL** is returned.

CONTEXT

esballoc( ) can be called from user or interrupt context.

SEE ALSO

allocb(9F), **freeb**(9F), **datab**(9S), **free_rtn**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

WARNINGS

The **free_func**( ) must be defined in kernel space, should be declared **void** and accept one argument. It has no user context and must not sleep.

NAME

esbbcall – call function when buffer is available

SYNOPSIS

#include <sys/stream.h>

int esbbcall(int *pri***, void (**∗*func***)(long arg)**,**long** *arg* **);**

ARGUMENTS

pri Priority of allocation request (to be used by **allocb**(9F) function, called by **esbbcall**( ))

func Function to be called when buffer becomes available.

arg Argument to *func*.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

esbbcall( ), like **bufcall**(9F), serves as a **timeout**(9F) call of indeterminate length. If **esballoc**(9F) is unable to allocate a message and data block header to go with its externally supplied data buffer, **esbbcall**( ) can be used to schedule the routine *func*, to be called with the argument *arg* when a buffer becomes available. *func* may be a routine that calls **esballoc** (9F) or it may be another kernel function.

RETURN VALUES

On success, a non-zero integer is returned. On failure, **0** is returned.
The value returned from a successful call should be saved for possible future use with **unbufcall**( ) should it become necessary to cancel the **esbbcall( )** request (as at driver close time).

CONTEXT

esbbcall( ) can be called from user or interrupt context.

SEE ALSO

allocb(9F), **bufcall**(9F), **esballoc**(9F), **timeout**(9F), **datab**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

flushband – flush messages for a specified priority band

SYNOPSIS

#include <sys/stream.h>

void flushband(queue_t **q*, **unsigned char** *pri*, **int** *flag***);**

ARGUMENTS

q Pointer to the queue.

pri Priority of messages to be flushed.

flag Valid *flag* values are:

- **FLUSHDATA** Flush only data messages (types **M_DATA, M_DELAY, M_PROTO,** and **M_PCPROTO).**
- **FLUSHALL** Flush all messages.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

flushband( ) flushes messages associated with the priority band specified by *pri*. If *pri* is **0**, only normal and high priority messages are flushed. Otherwise, messages are flushed from the band *pri* according to the value of *flag*.

CONTEXT

flushband( ) can be called from user or interrupt context.

SEE ALSO

flushq(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

flushq – remove messages from a queue

SYNOPSIS

#include <sys/stream.h>

void flushq(queue_t ∗*q*, **int** *flag***);**

ARGUMENTS

q Pointer to the queue to be flushed.

flag Valid *flag* values are:

FLUSHDATA	Flush only data messages (types **M_DATA, M_DELAY, M_PROTO,** and **M_PCPROTO).**
FLUSHALL	Flush all messages.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

flushq( ) frees messages and their associated data structures by calling **freemsg**(9F). If the queue's count falls below the low water mark and the queue was blocking an upstream service procedure, the nearest upstream service procedure is enabled.

CONTEXT

flushq( ) can be called from user or interrupt context.

EXAMPLE

This example depicts the canonical flushing code for STREAMS modules. The module has a write service procedure and potentially has messages on the queue. If it receives an **M_FLUSH** message, and if the **FLUSHR** bit is on in the first byte of the message (line 10), then the read queue is flushed (line 11). If the **FLUSHW** bit is on (line 12), then the write queue is flushed (line 13). Then the message is passed along to the next entity in the stream (line 14). See the example for **qreply**(9F) for the canonical flushing code for drivers.

```
 1 /*
 2  * Module write-side put procedure.
 3  */
 4 xxxwput(q, mp)
 5    queue_t *q;
 6    mblk_t *mp;
 7 {
 8        switch(mp->b_datap->db_type) {
 9          case M_FLUSH:
```

```
                if (*mp->b_rptr & FLUSHR)
                        flushq(RD(q), FLUSHALL);
                if (*mp->b_rptr & FLUSHW)
                        flushq(q, FLUSHALL);
                putnext(q, mp);
                break;
                . . .
        }
}
```

SEE ALSO

flushband(9F), **freemsg**(9F), **putq**(9F), **qreply**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

freeb – free a message block

SYNOPSIS

#include <sys/stream.h>

void freeb(mblk_t **bp*);**

ARGUMENTS

bp Pointer to the message block to be deallocated. **mblk_t** is an instance of the **msgb**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

freeb( ) deallocates a message block. If the reference count of the **db_ref** member of the **datab**(9S) structure is greater than **1**, **freeb**( ) decrements the count. If **db_ref** equals **1**, it deallocates the message block and the corresponding data block and buffer.

If the data buffer to be freed was allocated with the **esballoc**(9F), the buffer may be a non-STREAMS resource. In that case, the driver must be notified that the attached data buffer needs to be freed, and run its own freeing routine. To make this process independent of the driver used in the stream, **freeb**( ) finds the **free_rtn**(9S) structure associated with the buffer. The **free_rtn** structure contains a pointer to the driver-dependent routine, which releases the buffer. Once this is accomplished, **freeb**( ) releases the STREAMS resources associated with the buffer.

CONTEXT

freeb( ) can be called from user or interrupt context.

EXAMPLE

See **copyb**(9F) for an example of using **freeb**( ).

SEE ALSO

allocb(9F), **dupb**(9F), **esballoc**(9F), **free_rtn**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

freemsg – free all message blocks in a message

SYNOPSIS

#include <sys/stream.h>

void freemsg(mblk_t **mp***);**

ARGUMENTS

mp Pointer to the message blocks to be deallocated. **mblk_t** is an instance of the **msgb**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

freemsg( ) calls **freeb**(9F) to free all message and data blocks associated with the message pointed to by *mp*.

CONTEXT

freemsg( ) can be called from user or interrupt context.

EXAMPLE

See **copymsg**(9F).

SEE ALSO

freeb(9F), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

freerbuf – free a raw buffer header

SYNOPSIS

#include <sys/buf.h>
#include <sys/ddi.h>

void freerbuf(struct buf **bp***);**

ARGUMENTS

bp Pointer to a previously allocated buffer header structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

freerbuf( ) frees a raw buffer header previously allocated by **getrbuf**(9F). This function does not sleep and so may be called from an interrupt routine.

CONTEXT

freerbuf( ) can be called from user or interrupt context.

SEE ALSO

getrbuf(9F), **kmem_alloc**(9F), **kmem_free**(9F), **kmem_zalloc**(9F)

NAME

freezestr, unfreezestr – freeze, thaw the state of a stream

SYNOPSIS

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

void freezestr(queue_t **q***)***;*

void unfreezestr(queue_t **q***)***;*

ARGUMENTS

q Pointer to the message queue to freeze / unfreeze.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

freezestr( ) freezes the state of the entire stream containing the queue pair *q*. A frozen stream blocks any thread attempting to enter any open, close, put or service routine belonging to any queue instance in the stream, and blocks any thread currently within the stream if it attempts to put messages onto or take messages off of any queue within the stream (with the sole exception of the caller). Threads blocked by this mechanism remain so until the stream is thawed by a call to **unfreezestr( )**.

Drivers and modules must freeze the stream before manipulating the queues directly (as opposed to manipulating them through programmatic interfaces such as **getq**(9F), **putq**(9F), **putbq**(9F), etc.) They further must freeze the stream before accessing any queues through calls to **insq**(9F), **rmvq**(9F), **strqset**(9F) and **strqget**(9F).

CONTEXT

These routines may be called from any stream open, close, put or service routine as well as interrupt handlers, callouts and call-backs.

SEE ALSO

getq(9F), **insq**(9F), **putbq**(9F), **putq**(9F), **rmvq**(9F), **strqget**(9F), **strqset**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES

Calling **freezestr( )** to freeze a stream that is already frozen by the caller will result in a single-party deadlock.

The caller of **unfreezestr( )** must be the thread who called **freezestr( )**.

Global kernel locks and locks local to drivers and modules may be held across calls to these two routines. Beware of hierarchy violations with respect to local locks (locking policies established by the driver or module writer).

There are usually better ways to accomplish things than by freezing the stream.

STREAMS utility functions such as **getq**(9F), **putq**(9F), **putbq**(9F), etc. may not be called by the caller of **freezestr( )** while the stream is still frozen, as they indirectly freeze the stream to ensure atomicity of queue manipulation.

NAME

get_pktiopb, free_pktiopb – allocate / free a SCSI packet in the iopb map

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt *get_pktiopb(struct scsi_address **ap*, **caddr_t** **datap*, **int** *cdblen*, **int** *statuslen*, **int** *datalen*, **int** *readflag*, **int (****callback*)**(void));**

void free_pktiopb(struct scsi_pkt **pkt*, **caddr_t** *datap*, **int** *datalen*);

ARGUMENTS

ap Pointer to the target's **scsi_address** structure.

datap Pointer to the address of the packet, set by this function.

cdblen Number of bytes required for the SCSI command descriptor block (CDB).

statuslen Number of bytes required for the SCSI status area.

datalen Number of bytes required for the data area of the SCSI command.

readflag If non-zero, data will be transferred from the SCSI target.

callback Pointer to a callback function, or **NULL_FUNC** or **SLEEP_FUNC**

pkt Pointer to a **scsi_pkt**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

get_pktiopb( ) allocates a **scsi_pkt** structure that has a small data area allocated. It is used by some SCSI commands such as **REQUEST_SENSE**, which involve a small amount of data and require cache-consistent memory for proper operation. It uses **ddi_iopb_alloc**(9F) for allocating the data area and **scsi_resalloc**(9F) to allocate the packet and DMA resources.

callback indicates what **get_pktiopb( )** should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a NULL pointer.

SLEEP_FUNC Wait indefinitely for resources.

Other Values *callback* points to a function which is called when resources may have become available. *callback* **must** return either **0** (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or **1** indicating either success in allocating resources or indicating that it no longer cares for a retry.

free_pktiopb( ) is used for freeing the packet and its associated resources.

RETURN VALUES

get_pktiopb( ) returns a pointer to the newly allocated **scsi_pkt** or a **NULL** pointer.

CONTEXT

If *callback* is **SLEEP_FUNC**, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block.

free_pktiopb() can be called from user or interrupt context.

SEE ALSO

ddi_iopb_alloc(9F), **scsi_alloc_consistent_buf**(9F) **scsi_pktalloc**(9F), **scsi_resalloc**(9F), **scsi_pkt**(9S)

Writing Device Drivers

NOTES

get_pktiopb() and **free_pktiopb()** are old functions and should be replaced with **scsi_alloc_consistent_buf**(9F) and **scsi_free_consistent_buf**(9F). **get_pktiopb()** uses scarce resources. Use it selectively.

NAME

geterror – return I / O error

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>
#include <sys/ddi.h>

int geterror(struct buf *∗bp***);**

ARGUMENTS

bp Pointer to a **buf**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

geterror( ) returns the error number from the error field of the buffer header structure.

RETURN VALUES

An error number indicating the error condition of the I / O request is returned. If the I / O request completes successfully, **0** is returned.

CONTEXT

geterror( ) can be called from user or interrupt context.

SEE ALSO

buf(9S)

Writing Device Drivers

NAME

getmajor – get major device number

SYNOPSIS

#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>

major_t getmajor(dev_t *dev***);**

ARGUMENTS

dev Device number.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

getmajor( ) extracts the major number from a device number.

RETURN VALUES

The major number.

CONTEXT

getmajor( ) can be called from user or interrupt context.

EXAMPLE

The following example shows both the **getmajor**( ) and **getminor**(9F) functions used in a debug **cmn_err**(9F) statement to return the major and minor numbers for the device supported by the driver.

```
dev_t dev;

#ifdef DEBUG
cmn_err(CE_NOTE,"Driver Started.  Major# = %d,
        Minor# = %d", getmajor(dev), getminor(dev));
#endif
```

SEE ALSO

cmn_err(9F), **getminor**(9F), **makedevice**(9F)

Writing Device Drivers

WARNINGS

No validity checking is performed. If *dev* is invalid, an invalid number is returned.

NAME

getminor – get minor device number

SYNOPSIS

#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>

minor_t getminor(dev_t *dev***);**

ARGUMENTS

dev Device number.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

getminor( ) extracts the minor number from a device number.

RETURN VALUES

The minor number.

CONTEXT

getminor( ) can be called from user or interrupt context.

EXAMPLE

See the **getmajor**(9F) manual page for an example of how to use **getminor**(9F).

SEE ALSO

getmajor(9F), **makedevice**(9F)

Writing Device Drivers

WARNINGS

No validity checking is performed. If *dev* is invalid, an invalid number is returned.

NAME

getq – get the next message from a queue

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**getq(queue_t** ∗*q***);**

ARGUMENTS

q Pointer to the queue from which the message is to be retrieved.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

getq( ) is used by a service (**srv**(9E)) routine to retrieve its enqueued messages.

A module or driver may include a service routine to process enqueued messages. Once the STREAMS scheduler calls **srv**( ) it must process all enqueued messages, unless prevented by flow control. **getq**( ) obtains the next available message from the top of the queue pointed to by *q*. It should be called in a **while** loop that is exited only when there are no more messages or flow control prevents further processing.

If an attempt was made to write to the queue while it was blocked by flow control, **getq**( ) back-enables (restarts) the service routine once it falls below the low water mark.

RETURN VALUES

If there is a message to retrieve, **getq( )** returns a pointer to it. If no message is queued, **getq( )** returns a **NULL** pointer.

CONTEXT

getq( ) can be called from user or interrupt context.

EXAMPLE

See **dupb**(9F).

SEE ALSO

srv(9E), **bcanput**(9F), **canput**(9F), **putbq**(9F), **putq**(9F), **qenable**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

getrbuf – get a raw buffer header

SYNOPSIS

#include <sys/buf.h>
#include <sys/kmem.h>
#include <sys/ddi.h>

struct buf *getrbuf(long *sleepflag***);**

ARGUMENTS

sleepflag Indicates whether driver should sleep for free space.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

getrbuf() allocates the space for a buffer header to the caller. It is used in cases where a block driver is performing raw (character interface) I/O and needs to set up a buffer header that is not associated with the buffer cache.

getrbuf( ) calls **kmem_alloc**(9F) to perform the memory allocation. **kmem_alloc**( ) requires the information included in the *sleepflag* argument. If *sleepflag* is set to **KM_SLEEP**, the driver may sleep until the space is freed up. If *sleepflag* is set to **KM_NOSLEEP**, the driver will not sleep. In either case, a pointer to the allocated space is returned or **NULL** to indicate that no space was available.

RETURN VALUES

getrbuf() returns a pointer to the allocated buffer header, or **NULL** if no space is available.

CONTEXT

getrbuf() can be called from user or interrupt context. (Drivers must not allow **getrbuf()** to sleep if called from an interrupt routine.)

SEE ALSO

freerbuf(9F), **kmem_alloc**(9F), **kmem_free**(9F)

Writing Device Drivers

NAME

hat_getkpfnum – get page frame number for kernel address

SYNOPSIS

#include <sys/vm.h>
#include <sys/types.h>
#include <sys/ddi.h>

u_int hat_getkpfnum(caddr_t *addr***);**

ARGUMENTS

addr The kernel virtual address for which the page frame number is to be returned.

INTERFACE LEVEL

Architecture independent level 2 (DKI only).

DESCRIPTION

Drivers implementing the **mmap**(9E) entry point must return the page frame number corresponding to the virtual address of the device memory address *addr*, or –**1** for error. This frame number can be obtained by a call to **hat_getkpfnum()**.

RETURN VALUES

The page frame number corresponding to virtual address *addr*, or –1 for invalid mappings.

CONTEXT

hat_getkpfnum() can be called from user or interrupt context. Although there is no reason why **hat_getkpfnum()** cannot be called from interrupt context, there is no need, since it only needs to be called from within **mmap**(9E).

SEE ALSO

mmap(9E)

Writing Device Drivers

NAME

inb, inw, inl, repinsb, repinsw, repinsd – read from an I/O port

SYNOPSIS

#include <sys/ddi.h>

#include <sys/sunddi.h>

unsigned char inb(int *port***);**

unsigned short inw(int *port***);**

unsigned long inl(int *port***);**

void repinsb(int *port***, unsigned char** **addr***, int** *count***);**

void repinsw(int *port***, unsigned short** **addr***, int** *count***);**

void repinsd(int *port***, unsigned long** **addr***, int** *count***);**

ARGUMENTS

port A valid I/O port address.

addr The address of a buffer where the values will be stored.

count The number of values to be read from the I/O port.

INTERFACE LEVEL

Solaris x86 DDI specific (Solaris x86 DDI).

AVAILABILITY

x86

DESCRIPTION

These routines read data of various sizes from the I/O port with the address specified by *port*.

The **inb( )**, **inw( )**, and **inl( )** functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The **repinsb( )**, **repinsw( )**, and **repinsd( )** functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. *count* specifies the number of values to be read. A a pointer to a buffer will receive the input data; the buffer must be long enough to hold count values of the requested size.

RETURN VALUES

inb( ), **inw( )**, and **inl( )** return the value that was read from the I/O port.

CONTEXT

These functions may be called from user or interrupt context.

SEE ALSO

eisa(4), **isa**(4), **mca**(4), **outb**(9F)

Writing Device Drivers

NAME

insq – insert a message into a queue

SYNOPSIS

#include <sys/stream.h>

int insq(queue_t *∗q*, **mblk_t** *∗emp*, **mblk_t** *∗nmp***);**

ARGUMENTS

q — Pointer to the queue containing message *emp*.

emp — Enqueued message before which the new message is to be inserted. **mblk_t** is an instance of the **msgb**(9S) structure.

nmp — Message to be inserted.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

insq( ) inserts a message into a queue. The message to be inserted, *nmp*, is placed in *q* immediately before the message *emp*. If *emp* is **NULL**, the new message is placed at the end of the queue. The queue class of the new message is ignored. All flow control parameters are updated. The service procedure is enabled unless **QNOENB** is set.

RETURN VALUES

insq( ) returns **1** on success, and **0** on failure.

CONTEXT

insq( ) can be called from user or interrupt context.

EXAMPLE

This routine illustrates the steps a transport provider may take to place expedited data ahead of normal data on a queue (assume all **M_DATA** messages are converted into **M_PROTO T_DATA_REQ** messages). Normal **T_DATA_REQ** messages are just placed on the end of the queue (line 16). However, expedited **T_EXDATA_REQ** messages are inserted before any normal messages already on the queue (line 25). If there are no normal messages on the queue, **bp** will be **NULL** and we fall out of the **for** loop (line 21). **insq** acts like **putq**(9F) in this case.

```
 1 #include <sys/tihdr.h>
 2 #include <sys/stream.h>
 3
 4 static int
 5 xxxwput(queue_t *q, mblk_t *mp)
 6 {
 7         union T_primitives *tp;
 8         mblk_t *bp;
 9         union T_primitives *ntp;
10
11         switch (mp->b_datap->db_type) {
12         case M_PROTO:
13                 tp = (union T_primitives *)mp->b_rptr;
14                 switch (tp->type) {
15                 case T_DATA_REQ:
16                         putq(q, mp);
```

```
                        break;

                case T_EXDATA_REQ:
                        freezestr(q);
                        for (bp = q->q_first; bp; bp = bp->b_next) {
                          if (bp->b_datap->db_type == M_PROTO) {
                            ntp = (union T_primitives *)bp->b_rptr;
                            if (ntp->type != T_EXDATA_REQ)
                                break;
                          }
                        }
                        (void) insq(q, bp, mp);
                        unfreezestr(q);
                        break;
                . . .
        }
    }
}
```

SEE ALSO

freezestr(9F), **msgb**(9S), **putq**(9F), **unfreezestr**(9F), **rmvq**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

WARNINGS

If *emp* is non-NULL, it must point to a message on *q* or a system panic could result.

NOTES

The stream must be frozen using **freezestr**(9F) before calling **insq( ).**

NAME

kmem_alloc – allocate space from kernel free memory

SYNOPSIS

#include <sys/types.h>
#include <sys/kmem.h>

void ∗kmem_alloc(size_t *size***, int** *flag***);**

ARGUMENTS

size Number of bytes to allocate.

flag Determines if caller will sleep to wait for free space. Possible flags are **KM_SLEEP** to sleep while waiting for free space, and **KM_NOSLEEP** to return NULL if space is not available.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

kmem_alloc( ) allocates a specified amount of kernel memory in bytes and returns a pointer to the allocated memory. The *flag* argument determines whether the function will sleep while waiting for free space to be released. If *flag* has **KM_SLEEP** set, the caller may sleep until free space is available. If *flag* has **KM_NOSLEEP** set and space is not available, **NULL** will be returned.

RETURN VALUES

If successful, **kmem_alloc( )** returns a pointer to the allocated space. **NULL** is returned if **KM_NOSLEEP** is set and memory cannot be allocated.

CONTEXT

kmem_alloc( ) can be called from interrupt context only if the **KM_NOSLEEP** flag is set. It can be called from user context with any valid *flag*.

SEE ALSO

freerbuf(9F), **getrbuf**(9F), **kmem_free**(9F), **kmem_zalloc**(9F)

Writing Device Drivers

WARNINGS

Memory allocated by **kmem_alloc( )** is not paged. Available memory is therefore limited. Excessive use of this memory is likely to affect overall system performance.

NAME

kmem_free – free previously allocated kernel memory

SYNOPSIS

#include <sys/types.h>
#include <sys/kmem.h>

void kmem_free(void **cp*, **size_t** *size***);**

ARGUMENTS

cp Address of the allocated storage from which to return *size* of allocated memory.

size Number of bytes to free (same number of bytes as allocated by **kmem_alloc**(9F) or **kmem_zalloc**(9F).

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

kmem_free( ) returns *size* bytes of storage to kernel free space previously allocated by **kmem_alloc**(9F) or **kmem_zalloc**(9F). The *cp* and *size* values must specify exactly one complete area of allocated memory. One **kmem_free**( ) call must correspond to one allocation.

CONTEXT

kmem_free( ) can be called from user or interrupt context.

SEE ALSO

freerbuf(9F), **getrbuf**(9F), **kmem_alloc**(9F), **kmem_zalloc**(9F)

Writing Device Drivers

NAME

kmem_zalloc – allocate and clear space from kernel free memory

SYNOPSIS

#include <sys/types.h>
#include <sys/kmem.h>

void ∗kmem_zalloc(size_t *size***, int** *flags***);**

ARGUMENTS

size Number of bytes to allocate.

flags Determines if caller may sleep to wait for free space. Possible flags are **KM_SLEEP** to sleep while waiting for free space, and **KM_NOSLEEP** to return NULL if space is not available.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

This function allocates *size* bytes of storage from kernel free space, clears it, and returns a pointer to the allocated memory. If *flags* has **KM_SLEEP** set, the caller may sleep until free space is available. If *flags* has **KM_NOSLEEP** set and space is not available, **NULL** will be returned.

RETURN VALUES

kmem_zalloc( ) returns **NULL** if memory cannot be allocated. Otherwise, it returns a pointer to the allocated space.

CONTEXT

kmem_zalloc( ) can be called from interrupt context only if the **KM_NOSLEEP** flag is set. It can be called from user context with any valid *flags*.

SEE ALSO

freerbuf(9F), **getrbuf**(9F), **kmem_alloc**(9F), **kmem_free**(9F)

Writing Device Drivers

WARNINGS

Memory allocated by **kmem_zalloc**( ) is not paged. Available memory is therefore limited. Excessive use of this memory is likely to affect overall system performance.

NAME

kstat_create – create and initialize a new kstat

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

kstat_t ∗**kstat_create(char** ∗*module***, int** *instance***, char** ∗*name***, char** ∗*class* **, uchar_t** *type***, ulong_t** *ndata***, uchar_t** *ks_flag***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

module The name of the provider's module (such as "sd", "esp", ...). The "core" kernel (**/kernel/unix**) uses the name "unix".

instance The provider's instance number, as from **ddi_get_instance**(9F). Modules which don't have a meaningful instance number should use **0**.

name A pointer to a string that uniquely identifies this structure. Only **KSTAT_STRLEN - 1** characters are significant.

class The general class that this kstat belongs to. The following classes are currently in use: **disk**, **tape**, **net**, **controller**, **vm**, **kvm**, **hat**, **streams**, **kstat**, and **misc**.

type The type of kstat to allocate. Valid types are:

KSTAT_TYPE_NAMED named **-** allows more than one data record per kstat

KSTAT_TYPE_INTR interrupt **-** only one data record per kstat

KSTAT_TYPE_IO I / O **-** only one data record per kstat

ndata The number of type-specific data records to allocate.

flag A bit-field of various flags for this kstat. **flag** is some combination of:

KSTAT_FLAG_VIRTUAL Tells **kstat_create( )** not to allocate memory for the kstat data section; instead, the driver will set the **ks_data** field to point to the data it wishes to export. This provides a convenient way to export existing data structures.

KSTAT_FLAG_WRITABLE Makes the kstat's data section writable by root.

KSTAT_FLAG_PERSISTENT

Indicates that this kstat is to be persistent over time. For persistent kstats, **kstat_delete**(9F) simply marks the kstat as dormant; a subsequent **kstat_create( )** reactivates the kstat. This feature is provided so that statistics are not lost across driver close / open (such as raw disk I / O on a disk with no mounted partitions.)

Note: Persistent kstats cannot be virtual, since ks_data points to garbage as soon as the driver goes away.

DESCRIPTION

kstat_create( ) is used in conjunction with **kstat_install**(9F) to allocate and initialize a **kstat**(9S) structure. The method is generally as follows:

```
kstat_t  *ksp;

ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
        /* ... provider initialization, if necessary */
        kstat_install(ksp);
}
```

kstat_create( ) allocates and performs necessary system initialization of a **kstat**(9S) structure. **kstat_create( )** allocates memory for the entire kstat (header plus data), initializes all header fields, initializes the data section to all zeroes, assigns a unique kstat ID (KID), and puts the kstat onto the system's kstat chain. The returned kstat is marked invalid because the provider (caller) has not yet had a chance to initialize the data section.

After a successful call to **kstat_create( )** the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type **KSTAT_TYPE_NAMED**). Virtual kstats must have the **ks_data** field set at this time. The provider may also set the **ks_update**, **ks_private**, and **ks_lock** fields if necessary.

Once the kstat is completely initialized, **kstat_install**(9F) is used to make the kstat accessible to the outside world.

RETURN VALUES

If successful, **kstat_create( )** returns a pointer to the allocated kstat. **NULL** is returned on failure.

CONTEXT

kstat_create( ) can be called from user or kernel context.

SEE ALSO

kstat(3K), **kstat_delete**(9F), **kstat_install**(9F), **kstat_named_init**(9F), **kstat**(9S), **kstat_named**(9S)

Writing Device Drivers

NAME

kstat_delete – remove a kstat from the system

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

void kstat_delete(kstat_t ∗*ksp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

ksp Pointer to a currently installed **kstat**(9S) structure.

DESCRIPTION

kstat_delete( ) removes ksp from the kstat chain and frees all associated system resources.

RETURN VALUES

None.

CONTEXT

kstat_delete( ) can be called from any context.

SEE ALSO

kstat_create(9F), **kstat_install**(9F), **kstat_named_init**(9F), **kstat**(9S)

Writing Device Drivers

NOTES

When calling **kstat_delete( )**, the driver must *not* be holding that kstat's **ks_lock**. Otherwise, it may deadlock with a kstat reader.

NAME

kstat_install – add a fully initialized kstat to the system

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

void kstat_install(kstat_t ∗*ksp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

ksp Pointer to a fully initialized **kstat**(9S) structure.

DESCRIPTION

kstat_install( ) is used in conjunction with **kstat_create**(9F) to allocate and initialize a **kstat**(9S) structure. The method is generally as follows:

```
kstat_t *ksp;

ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
        /* ... provider initialization, if necessary */
        kstat_install(ksp);
}
```

After a successful call to **kstat_create( )** the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type **KSTAT_TYPE_NAMED**). Virtual kstats must have the **ks_data** field set at this time. The provider may also set the **ks_update**, **ks_private**, and **ks_lock** fields if necessary.

Once the kstat is completely initialized, **kstat_install**(9F) is used to make the kstat accessible to the outside world.

RETURN VALUES

None.

CONTEXT

kstat_install( ) can be called from user or kernel context.

SEE ALSO

kstat_create(9F), **kstat_delete**(9F), **kstat_named_init**(9F), **kstat**(9S)

Writing Device Drivers

NAME

kstat_named_init – initialize a named kstat

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

void kstat_named_init(kstat_named_t **knp*, **char** **name*, **uchar_t** *data_type***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

knp Pointer to a **kstat_named**(9S) structure.

name The name of the statistic.

data_type The type of value. This indicates which field of the **kstat_named**(9S) structure should be used. Valid values are:

KSTAT_DATA_CHAR	the "char" field.
KSTAT_DATA_LONG	the "long" field.
KSTAT_DATA_ULONG	the "unsigned long" field.
KSTAT_DATA_LONGLONG	the "long long" field.
KSTAT_DATA_ULONGLONG	the "unsigned long long" field.

DESCRIPTION

kstat_named_init( ) associates a name and a type with a **kstat_named**(9S) structure.

RETURN VALUES

None.

CONTEXT

kstat_named_init( ) can be called from user or kernel context.

SEE ALSO

kstat_create(9F), **kstat_install**(9F), **kstat**(9S), **kstat_named**(9S)

Writing Device Drivers

NAME

kstat_queue, kstat_waitq_enter, kstat_waitq_exit, kstat_runq_enter, kstat_runq_exit, kstat_waitq_to_runq, kstat_runq_back_to_waitq – update I/O kstat statistics

SYNOPSIS

#include <sys/types.h>
#include <sys/kstat.h>

void kstat_waitq_enter(kstat_io_t **kiop***);**

void kstat_waitq_exit(kstat_io_t **kiop***);**

void kstat_runq_enter(kstat_io_t **kiop***);**

void kstat_runq_exit(kstat_io_t **kiop***);**

void kstat_waitq_to_runq(kstat_io_t **kiop***);**

void kstat_runq_back_to_waitq(kstat_io_t **kiop***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI)

ARGUMENTS

kiop Pointer to a **kstat_io**(9S) structure.

DESCRIPTION

A large number of I/O subsystems have at least two basic "lists" (or queues) of transactions they manage: one for transactions that have been accepted for processing but for which processing has yet to begin, and one for transactions which are actively being processed (but not done). For this reason, two cumulative time statistics are kept: wait (pre-service) time, and run (service) time.

The **kstat_queue( )** family of functions manage these times based on the transitions between the driver wait queue and run queue.

kstat_waitq_enter( )

kstat_waitq_enter( ) should be called when a request arrives and is placed into a pre-service state (such as just prior to calling **disksort**(9F)).

kstat_waitq_exit( )

kstat_waitq_exit( ) should be used when a request is removed from its pre-service state. (such as just prior to calling the driver's **start** routine).

kstat_runq_enter( )

kstat_runq_enter( ) is also called when a request is placed in its service state (just prior to calling the driver's start routine, but after **kstat_waitq_exit( )**).

kstat_runq_exit( )

kstat_runq_exit( ) is used when a request is removed from its service state (just prior to calling **biodone**(9F)).

kstat_waitq_to_runq( )

kstat_waitq_to_runq( ) transitions a request from the wait queue to the run queue. This is useful wherever the driver would have normally done a **kstat_waitq_exit( )** followed by a call to **kstat_runq_enter( )**.

kstat_runq_back_to_waitq( )

kstat_runq_back_to_waitq( ) transitions a request from the run queue back to the wait queue. This may be necessary in some cases (write throttling is an example).

RETURN VALUES

None.

CONTEXT

kstat_create( ) can be called from user or kernel context.

WARNINGS

These transitions must be protected by holding the kstat's **ks_lock**, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.

SEE ALSO

kstat_create(9F), **kstat_delete**(9F), **kstat_named_init**(9F), **kstat**(9S), **kstat_io**(9S)

Writing Device Drivers

NAME

linkb – concatenate two message blocks

SYNOPSIS

#include <sys/stream.h>

void linkb(mblk_t ∗*mp1***, mblk_t** ∗*mp2***);**

ARGUMENTS

mp1 The message to which *mp2* is to be added. **mblk_t** is an instance of the **msgb**(9S) structure.

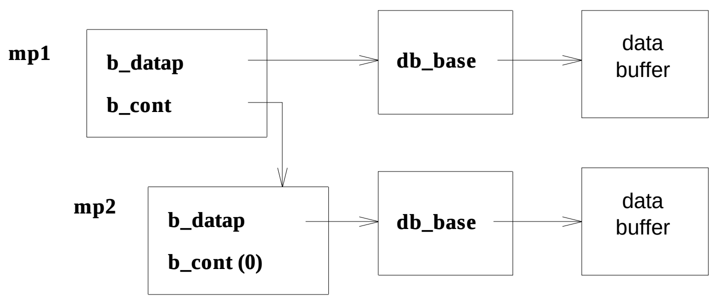
mp2 The message to be added.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

linkb( ) creates a new message by adding *mp2* to the tail of *mp1*. The continuation pointer, **b_cont**, of the first message is set to point to the second message:



linkb(mp1, mp2);

CONTEXT

linkb( ) can be called from user or interrupt context.

EXAMPLE

See **dupb**(9F) for an example of using **linkb**( ).

SEE ALSO

unlinkb(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

makecom, makecom_g0, makecom_g0_s, makecom_g1, makecom_g5 – make a packet for SCSI commands

SYNOPSIS

#include <sys/scsi/scsi.h>

void makecom_g0(struct scsi_pkt **pkt,* **struct scsi_device** **devp***, int** *flag,* **int** *cmd,* **int** *addr,* **int** *cnt***);**

void makecom_g0_s(struct scsi_pkt **pkt,* **struct scsi_device** **devp,* **int** *flag,* **int** *cmd,* **int** *cnt,* **int** *fixbit***);**

void makecom_g1(struct scsi_pkt **pkt,* **struct scsi_device** **devp,* **int** *flag,* **int** *cmd,* **int** *addr,* **int** *cnt***);**

void makecom_g5(struct scsi_pkt **pkt,* **struct scsi_device** **devp,* **int** *flag,* **int** *cmd,* **int** *addr,* **int** *cnt***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

pkt Pointer to an allocated **scsi_pkt**(9S) structure.

devp Pointer to the target's **scsi_device**(9S) structure.

flag Flags for the pkt_flags variable.

cmd The SCSI Group 0 or 1 or 5 command.

addr Pointer to the location of the data.

cnt Number of bytes to transfer.

fixbit Fixed bit in sequential access device commands.

DESCRIPTION

makecom functions initialize a packet with the specified command descriptor block, *devp* and transport flags. The *pkt_address*, *pkt_flags*, and the command descriptor block pointed to by *pkt_cdbp* are initialized using the remaining arguments. Target drivers may use **makecom_g0( )** for Group 0 commands (except for sequential access devices), or **makecom_g0_s( )** for Group 0 commands for sequential access devices, or **makecom_g1( )** for Group 1 commands, or **makecom_g5( )** for Group 5 commands. *fixbit* is used by sequential access devices for accessing fixed block sizes and sets the the tag portion of the SCSI CDB.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLE

```
if (blkno >= (1<<20)) {
        makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1,
            (int) blkno, nblk);
} else {
        makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE,
            (int) blkno, nblk);
}
```

SEE ALSO **scsi_pkt**(9S)

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

Writing Device Drivers

NAME

makedevice – make device number from major and minor numbers

SYNOPSIS

#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>

dev_t makedevice(major_t *majnum***, minor_t** *minnum***);**

ARGUMENTS

majnum Major device number.

minnum Minor device number.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

makedevice( ) creates a device number from a major and minor device number. **makedevice**( ) should be used to create device numbers so the driver will port easily to releases that treat device numbers differently.

RETURN VALUES

The device number, containing both the major number and the minor number, is returned. No validation of the major or minor numbers is performed.

CONTEXT

makedevice( ) can be called from user or interrupt context.

SEE ALSO

getmajor(9F), **getminor**(9F)

NAME

max – return the larger of two integers

SYNOPSIS

#include <sys/ddi.h>

int max(int *int1*, **int** *int2***);**

ARGUMENTS

int1 The first integer.

int2 The second integer.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

max( ) compares two signed integers and returns the larger of the two.

RETURN VALUES

The larger of the two numbers.

CONTEXT

max( ) can be called from user or interrupt context.

SEE ALSO

min(9F)

Writing Device Drivers

NAME

min – return the lesser of two integers

SYNOPSIS

#include <sys/ddi.h>

int min(int *int1***, int** *int2***);**

ARGUMENTS

int1 The first integer.

int2 The second integer.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

min( ) compares two signed integers and returns the lesser of the two.

RETURN VALUES

The lesser of the two integers.

CONTEXT

min( ) can be called from user or interrupt context.

SEE ALSO

max(9F)

Writing Device Drivers

NAME

mod_install, mod_remove, mod_info – add, remove or query a loadable module

SYNOPSIS

#include <sys/modctl.h>

int mod_install(struct modlinkage **modlinkage);*

int mod_remove(struct modlinkage **modlinkage);*

int mod_info(struct modlinkage **modlinkage*, **struct modinfo** **modinfo*);**

ARGUMENTS

modlinkage Pointer to the loadable module's modlinkage structure which describes what type(s) of module elements are included in this loadable module.

modinfo Pointer to the **modinfo** structure passed to **_info**(9E).

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

mod_install( ) must be called from a module's **_init**(9E) routine.

mod_remove( ) must be called from a module's **_fini**(9E) routine.

mod_info( ) must be called from a module's **_info**(9E) routine.

RETURN VALUES

These functions all return zero on success and non-zero on failure.

EXAMPLES

For an example of using these functions see **_init**(9E).

SEE ALSO

_fini(9E), **_info**(9E), **_init**(9E), **modldrv**(9S), **modlinkage**(9S), **modlstrmod**(9S)

Writing Device Drivers

NAME

msgdsize – return the number of bytes in a message

SYNOPSIS

#include <sys/stream.h>

int msgdsize(mblk_t **mp***);**

ARGUMENTS

mp Message to be evaluated.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

msgdsize( ) counts the number of bytes in a data message. Only bytes included in the data blocks of type **M_DATA** are included in the count.

RETURN VALUES

The number of data bytes in a message, expressed as an integer.

CONTEXT

msgdsize( ) can be called from user or interrupt context.

EXAMPLE

See **bufcall**(9F) for an example of using **msgdsize**( ).

SEE ALSO

bufcall(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

msgpullup – concatenate bytes in a message

SYNOPSIS

#include <sys/stream.h>

mblk_t ∗**msgpullup(mblk_t** ∗*mp***, int** *len***);**

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

ARGUMENTS

mp Pointer to the message whose blocks are to be concatenated.

len Number of bytes to concatenate.

DESCRIPTION

msgpullup( ) concatenates and aligns the first *len* data bytes of the message pointed to by *mp*, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered. If *len* equals –**1**, all data are concatenated. If *len* bytes of the same message type cannot be found, **msgpullup( )** fails and returns **NULL.**

RETURN VALUES

On success, a pointer to the new message is returned; on failure, **NULL** is returned.

CONTEXT

msgpullup( ) can be called from user or interrupt context.

SEE ALSO

srv(9E), **allocb**(9F), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES

msgpullup( ) is a DKI-complaint replacement for the older **pullupmsg**(9F) routine. Users are strongly encouraged to use **msgpullup( )** instead of of **pullupmsg**(9F).

NAME mt-streams – STREAMS multithreading

SYNOPSIS **#include <sys/conf.h>**

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

DESCRIPTION

STREAMS drivers configures the degree of concurrency using the **cb_flag** field in the **cb_ops** structure (see **cb_ops**(9S)). The corresponding field for STREAMS modules is the **f_flag** in the **fmodsw** structure.

For the purpose of restricting and controlling the concurrency in drivers / modules, we define the concepts of **inner** and **outer perimeters**. A driver / module can be configured either to have no perimeters, to have only an inner or an outer perimeter, or to have both an inner and an outer perimeter. Each perimeter acts as a readers-writers lock, that is, there can be multiple concurrent readers or a single writer. Thus, each perimeter can be entered in two modes: shared (reader) or exclusive (writer). The mode depends on the perimeter configuration and can be different for the different STREAMS entry points (**open**(9E), **close**(9E), **put**(9E), or **srv**(9E)).

The concurrency for the different entry points is (unless specified otherwise) to enter with exclusive access at the inner perimeter (if present) and shared access at the outer perimeter (if present).

The perimeter configuration consists of flags that define the presence and scope of the inner perimeter, the presence of the outer perimeter (which can only have one scope), and flags that modify the default concurrency for the different entry points.

All MT safe modules / drivers specify the D_MP flag.

Inner Perimeter Flags

The inner perimeter presence and scope are controlled by the mutually exclusive flags:

D_MTPERQ The module / driver has an inner perimeter around each queue.

D_MTQPAIR The module / driver has an inner perimeter around each read / write pair of queues.

D_MTPERMOD The module / driver has an inner perimeter that encloses all the module's / driver's queues.

None of the above The module / driver has no inner perimeter.

Outer Perimeter Flags

The outer perimeter presence is configured using:

D_MTOUTPERIM In addition to any inner perimeter, the module / driver has an outer perimeter that encloses all the module's / driver's queues. This can be combined with all the inner perimeter options except *D_MTPERMOD.*

The default concurrency can be modified using:

D_MTPUTSHARED This flag modifies the default behavior when **put**(9E) procedure are invoked so that the inner perimeter is entered shared instead of exclusively.

D_MTOCEXCL This flag modifies the default behavior when **open**(9E) and **close**(9E) procedures are invoked so the the outer perimeter is entered exclusively instead of shared.

The module / driver can use **qwait**(9F) or **qwait_sig( )** in the **open**(9E) and **close**(9E) procedures if it needs to wait "outside" the perimeters.

The module / driver can use **qwriter**(9F) to upgrade the access at the inner or outer perimeter from shared to exclusive.

The use and semantics of **qprocson( )** and **qprocsoff**(9F) is independent of the inner and outer perimeters.

SEE ALSO

cb_ops(9S), **qwait**(9F), **qwriter**(9F), **qprocson**(9F)

STREAMS Programmer's Guide

Writing Device Drivers

NAME

mutex, mutex_enter, mutex_exit, mutex_init, mutex_destroy, mutex_owned, mutex_tryenter – mutual exclusion lock routines

SYNOPSIS

#include <sys/ksynch.h>

void mutex_init(kmutex_t **mp*, **char** **name*, **kmutex_type_t** *type*, **void** **arg***)**

void mutex_destroy(kmutex_t **mp***)**

void mutex_enter(kmutex_t **mp***)**

void mutex_exit(kmutex_t **mp***)**

int mutex_owned(kmutex_t **mp***)**

int mutex_tryenter(kmutex_t **mp***)**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

mp	Pointer to a kernel mutex lock (**kmutex_t**).
name	Character string describing lock for statistics and debugging.
type	Type of mutex lock.
arg	Type-specific argument for initialization routine.

DESCRIPTION

A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked. That is to say, mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex_init() is used to initialize a mutex so that it is unlocked and has a particular variant type. The only DDI-compliant types provided are **MUTEX_DRIVER, MUTEX_DRIVER_NOSTAT,** and **MUTEX_DRIVER_STAT.** Most of the time, the type **MUTEX_DRIVER** should be used.

If the call is compiled with **_LOCKTEST** or **_MPSTATS** defined, statistics will be kept for **MUTEX_DRIVER** mutexes. Statistics are always maintained for type **MUTEX_DRIVER_STAT,** and never maintained for **MUTEX_DRIVER_NOSTAT.** Note that statistics may incur a performance penalty. In addition, the system may need to allocate memory associated with the mutex, depending on the type.

arg provides type-specific information for a given variant type of mutex. When **mutex_init()** is called for driver mutexes, the arg should be the **ddi_iblock_cookie** returned from **ddi_add_intr**(9F) if the mutex is used by the interrupt handler. If the mutex is never used inside an interrupt handler, the argument should be NULL.

mutex_enter() is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic will ensue.

mutex_owned( ) should only be used in ASSERTs, and may be enforced by not being defined unless the preprocessor symbol DEBUG is defined. Its return value is non-zero if the current thread (or, if that cannot be determined, at least some thread) holds the mutex pointed to by *mp*.

mutex_tryenter( ) is very similar to **mutex_enter( )** except that it doesn't block when the mutex is already held. **mutex_tryenter( )** returns non-zero when it acquired the mutex and 0 when the mutex is already held.

mutex_exit( ) releases a mutex and will unblock another thread if any are blocked on the mutex.

mutex_destroy( ) frees any storage associated with the mutex, which may have been allocated when **mutex_init( )** was called. This should be called before deallocating storage containing the mutex. The caller must somehow be sure that no other thread will attempt to use the mutex.

RETURN VALUES

mutex_tryenter( ) returns non-zero on success and zero of failure.

mutex_owned( ) returns non-zero if the calling thread currently holds the mutex pointed to by *mp*, or when that cannot be determined, if any thread holds the mutex. **mutex_owned( )** returns zero otherwise.

CONTEXT

These functions can be called from user or interrupt context, except for **mutex_init( )** and **mutex_destroy( )**, which can be called from user context only.

EXAMPLES

Initialization

A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

```
ddi_add_intr(dip, 0, &iblock, &dev_cookie, xxintr,
        (caddr_t)un);
mutex_init(&un->un_lock, "xx unit lock", MUTEX_DRIVER,
        (void *)iblock);
```

Also, a routine that expects to be called with a certain lock held might have the following ASSERT:

```
xxstart(struct xxunit *un)
{
        ASSERT(mutex_owned(&un->un_lock));
...
```

SEE ALSO

ddi_add_intr(9F), **condvar**(9F), **rwlock**(9F), **semaphore**(9F)

Writing Device Drivers

BUGS

There is currently no product support for looking at lock statistics.

NAME

nochpoll – error return function for non-pollable devices.

SYNOPSIS

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

int nochpoll(dev_t *dev,* **short** *events,* **int** *anyyet,* **short** **reventsp,*
struct pollhead ***pollhdrp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

dev	Device number.
events	Event flags.
anyyet	Check current events only.
reventsp	Event flag pointer.
pollhdrp	Poll head pointer.

DESCRIPTION

nochpoll( ) is a routine that simply returns the value **ENXIO**. It is intended to be used in the **cb_ops**(9S) structure of a device driver for devices that do not support the **poll**(2) system call.

RETURN VALUE

nochpoll( ) returns **ENXIO**.

CONTEXT

nochpoll( ) can be called from user or interrupt context.

SEE ALSO

poll(2), **cb_ops**(9S)

Writing Device Drivers

NAME

nodev – error return function

SYNOPSIS

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

int nodev( );

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

nodev( ) returns **ENXIO**. It is intended to be used in the **cb_ops**(9S) data structure of a device driver for device entry points which are not supported by the driver. That is, it is an error to attempt to call such an entry point.

RETURN VALUES

nodev( ) returns **ENXIO**.

CONTEXT

nodev( ) can be only called from user context.

SEE ALSO

nulldev(9F), **cb_ops**(9S)

Writing Device Drivers

NAME

noenable – prevent a queue from being scheduled

SYNOPSIS

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

void noenable(queue_t **q***);**

ARGUMENTS

q Pointer to the queue.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

noenable( ) prevents the queue *q* from being scheduled for service by **insq**(9F), **putq**(9F) or **putbq**(9F) when enqueuing an ordinary priority message. The queue can be re-enabled with the **enableok**(9F) function.

CONTEXT

noenable( ) can be called from user or interrupt context.

SEE ALSO

enableok(9F), **insq**(9F), **putbq**(9F), **putq**(9F), **qenable**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

nulldev – zero return function

SYNOPSIS

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

int nulldev( );

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

nulldev( ) returns **0**. It is intended to be used in the **cb_ops**(9S) data structure of a device driver for device entry points that do nothing.

RETURN VALUES

nulldev( ) returns a **0**.

CONTEXT

nulldev( ) can be called from any context.

SEE ALSO

nodev(9F), **cb_ops**(9S)

Writing Device Drivers

NAME

outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

SYNOPSIS

#include <sys/ddi.h>

#include <sys/sunddi.h>

void outb(int *port*, **unsigned char** *value***);**

void outw(int *port*, **unsigned short** *value***);**

void outl(int *port*, **unsigned long** *value***);**

void repoutsb(int *port*, **unsigned char** **addr*, **int** *count***);**

void repoutsw(int *port*, **unsigned short** **addr*, **int** *count***);**

void repoutsd(int *port*, **unsigned long** **addr*, **int** *count***);**

ARGUMENTS

port	A valid I/O port address.
value	The data to be written to the I/O port.
addr	The address of a buffer from which the values will be fetched.
count	The number of values to be written to the I/O port.

INTERFACE LEVEL

Solaris x86 DDI specific (Solaris x86 DDI).

AVAILABILITY

x86

DESCRIPTION

These routines write data of various sizes to the I/O port with the address specified by *port*.

The **outb( )**, **outw( )**, and **outl( )** functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by *value*.

The **repoutsb( )**, **repoutsw( )**, and **repoutsd( )** functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. *count* specifies the number of values to be written. *addr* is a pointer to a buffer from which the output values are fetched.

CONTEXT

These functions may be called from user or interrupt context.

SEE ALSO

eisa(4), **isa**(4), **mca**(4), **inb**(9F)

Writing Device Drivers

NAME

physio, minphys – perform physical I / O

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>
#include <sys/uio.h>

int physio(int *(*strat)***(struct buf *), struct buf** **bp***, dev_t** *dev*,
int *rw***, void** *(*mincnt)***(struct buf *), struct uio** **uio***);**

void minphys(struct buf **bp***);**

ARGUMENTS

physio()

strat Pointer to device strategy routine.

bp Pointer to a **buf**(9S) structure describing the transfer. If *bp* is set to **NULL** then **physio**( ) allocates one which is automatically released upon completion.

dev The device number.

rw Read / write flag. This is either B_READ when reading from the device, or B_WRITE when writing to the device.

mincnt Routine which bounds the maximum transfer unit size.

uio Pointer to the **uio** structure which describes the user I / O request.

minphys( )

bp Pointer to a **buf** structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

physio( ) performs unbuffered I / O operations between the device *dev* and the address space described in the **uio** structure.

Prior to the start of the transfer **physio( )** verifies the requested operation is valid by checking the protection of the address space specified in the **uio** structure. It then locks the pages involved in the I / O transfer so they can not be paged out. The device strategy routine, **strat**( ), is then called one or more times to perform the physical I / O operations. **physio( )** uses **biowait**(9F) to block until **strat**( ) has completed each transfer. Upon completion, or detection of an error, **physio( )** unlocks the pages and returns the error status.

physio( ) uses **mincnt**( ) to bound the maximum transfer unit size to the system, or device, maximum length. **minphys**( ) is the system **mincnt**( ) routine for use with **physio( )** operations. Drivers which do not provide their own local **mincnt( )** routines should call **physio( )** with **minphys( )**.

minphys( ) limits the value of **bp->b_bcount** to a sensible default for the capabilities of the system. Drivers that provide their own **mincnt**( ) routine should also call **minphys( )** to make sure they do not exceed the system limit.

RETURN VALUES

physio( ) returns:

0	on success.
non-zero	on failure.

CONTEXT

physio() can be called from user context only.

SEE ALSO

strategy(9E), **biodone**(9F), **biowait**(9F), **buf**(9S), **uio**(9S)

Writing Device Drivers

WARNINGS

Since **physio( )** calls **biowait**( ) to block until each buf transfer is complete, it is the drivers responsibility to call **biodone**(9F) when the transfer is complete, or **physio( )** will block forever.

NAME

pollwakeup – inform a process that an event has occurred

SYNOPSIS

#include <sys/poll.h>

void pollwakeup(struct pollhead **php*, **short** *event***);**

ARGUMENTS

php Pointer to a **pollhead** structure.

event Event to notify the process about.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

pollwakeup( ) wakes a process waiting on the occurrence of an event. It should be called from a driver for each occurrence of an event. The **pollhead** structure will usually be associated with the driver's private data structure associated with the particular minor device where the event has occurred. See **chpoll**(9E) and **poll**(2) for more detail.

CONTEXT

pollwakeup( ) can be called from user or interrupt context.

SEE ALSO

poll(2), **chpoll**(9E)

Writing Device Drivers

NOTES

Driver defined locks should not be held across calls to this function.

NAME

proc_signal, proc_ref, proc_unref – send a signal to a process

SYNOPSIS

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

void *proc_ref(void);

void proc_unref(void **pref*);**

int proc_signal(void **pref*, **int** *sig*);

ARGUMENTS

pref A handle for the process to be signalled.

sig Signal number to be sent to the process.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

This set of routines allows a driver to send a signal to a process. The routine **proc_ref( )** is used to retrieve an unambiguous reference to the process for signalling purposes. The return value can be used as a unique handle on the process, even if the process dies. Because system resources are committed to a process reference, **proc_unref( )** should be used to remove it as soon as it is no longer needed.

proc_signal( ) is used to send signal *sig* to the referenced process. The following set of signals may be sent to a process from a driver:

SIGHUP	The device has been disconnected
SIGINT	The interrupt character has been received
SIGQUIT	The quit character has been received
SIGPOLL	A pollable event has occurred.
SIGKILL	Kill the process (cannot be caught or ignored)
SIGWINCH	Window size change.
SIGURG	Urgent data are available.

See **signal**(5) for more details on the meaning of these signals.

If the process has exited at the time the signal was sent, **proc_signal( )** returns an error code; the caller should remove the reference on the process by calling **proc_unref( )**.

RETURN VALUES

proc_ref( )

pref An opaque handle used to refer to the current process.

proc_signal( )

0 The process existed before the signal was sent.

–1 The process no longer exists; no signal was sent.

CONTEXT

proc_unref() and **proc_signal()** can be called from user or interrupt context. **proc_ref()** should only be called from user context.

SEE ALSO

signal(5), **putnextctl1**(9F)

Writing Device Drivers

NAME

ptob – convert size in pages to size in bytes

SYNOPSIS

#include <sys/ddi.h>

unsigned long ptob(unsigned long *numpages***);**

ARGUMENTS

numpages Size in number of pages to convert to size in bytes.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

This function returns the number of bytes that are contained in the specified number of pages. For example, if the page size is 2048, then **ptob(2)** returns **4096**. **ptob(0)** returns **0**.

RETURN VALUES

The return value is always the number of bytes in the specified number of pages. There are no invalid input values, and no checking will be performed for overflow in the case of a page count whose corresponding byte count cannot be represented by an **unsigned long**. Rather, the higher order bits will be ignored.

CONTEXT

ptob( ) can be called from user or interrupt context.

SEE ALSO

btop(9F), **btopr**(9F), **ddi_ptob**(9F)

Writing Device Drivers

NAME

pullupmsg – concatenate bytes in a message

SYNOPSIS

#include <sys/stream.h>

int pullupmsg(mblk_t **mp*, **int** *len***);**

ARGUMENTS

mp Pointer to the message whose blocks are to be concatenated. **mblk_t** is an instance of the **msgb**(9S) structure.

len Number of bytes to concatenate.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

pullupmsg( ) tries to combine multiple data blocks into a single block. **pullupmsg( )** concatenates and aligns the first *len* data bytes of the message pointed to by *mp*. If *len* equals **-1**, all data are concatenated. If *len* bytes of the same message type cannot be found, **pullupmsg( )** fails and returns **0**.

RETURN VALUES

On success, **1** is returned; on failure, **0** is returned.

CONTEXT

pullupmsg( ) can be called from user or interrupt context.

EXAMPLE

This is a driver write **srv**(9E) (service) routine for a device that does not support scatter / gather DMA. For all **M_DATA** messages, the data will be transferred to the device with DMA.

First, try to pull up the message into one message block with the **pullupmsg( )** function (line 12). If successful, the transfer can be accomplished in one DMA job. Otherwise, it must be done one message block at a time (lines 19–22). After the data has been transferred to the device, free the message and continue processing messages on the queue.

```
 1 xxxwsrv(q)
 2     queue_t *q;
 3 {
 4         mblk_t *mp;
 5         mblk_t *tmp;
 6         caddr_t dma_addr;
 7         int dma_len;
 8
 9         while ((mp = getq(q)) != NULL) {
10                 switch (mp->b_datap->db_type) {
11                 case M_DATA:
12                         if (pullupmsg(mp, -1)) {
13                                 dma_addr = vtop(mp->b_rptr);
14                                 dma_len = mp->b_wptr - mp->b_rptr;
15                                 xxx_do_dma(dma_addr, dma_len);
16                                 freemsg(mp);
```

```
                        break;
                    }
                    for (tmp = mp; tmp; tmp = tmp->b_cont) {
                        dma_addr = vtop(tmp->b_rptr);
                        dma_len = tmp->b_wptr - tmp->b_rptr;
                        xxx_do_dma(dma_addr, dma_len);
                    }
                    freemsg(mp);
                    break;
        . . .
            }
        }
}
```

SEE ALSO **srv**(9E), **allocb**(9F), **msgpullup**(9F), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES **pullupmsg( )** is not included in the DKI and will be removed from the system in a future release. Device driver writers are strongly encouraged to use **msgpullup**(9F) instead of **pullupmsg( )**.

NAME

put – call a STREAMS put procedure

SYNOPSIS

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

void put(queue_t **q*, **mblk_t** **mp***);**

ARGUMENTS

q Pointer to a STREAMS queue.

mp Pointer to message block being passed into queue.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

put calls the put procedure (**put**(9E) entry point) for the STREAMS queue specified by *q*, passing it the message block referred to by *mp*. It is typically used by a driver or module to call its own put procedure.

CONTEXT

put can be called from a STREAMS module or driver put or service routine, or from an associated interrupt handler, timeout, bufcall, or esballoc call-back. In the latter cases the calling code must guarantee the validity of the *q* argument.

Since *put* may cause re-entry of the module (as it is intended to do), mutexes or other locks should not be held across calls to it, due to the risk of single-party deadlock.

NOTES

The caller cannot have the stream frozen (see **freezestr**(9F)) when calling this function.

DDI / DKI conforming modules and drivers are no longer permitted to call put procedures directly, but must call through the appropriate STREAMS utility function (e.g. **put**(9E), **putnext**(9F), **putctl**(9F), **qreply**(9F), etc). This function is provided as a DDI / DKI conforming replacement for a direct call to a put procedure.

SEE ALSO

put(9E), **putctl**(9F), **putctl1**(9F), **putnext**(9F), **putnextctl**(9F), **putnextctl1**(9F), **qreply**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putbq – place a message at the head of a queue

SYNOPSIS

#include <sys/stream.h>

int putbq(queue_t **q*, **mblk_t** **bp***);**

ARGUMENTS

q Pointer to the queue.

bp Pointer to the message block.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putbq( ) places a message at the beginning of the appropriate section of the message queue. There are always sections for high priority and ordinary messages. If other priority bands are used, each will have its own section of the queue, in priority band order, after high priority messages and before ordinary messages. **putbq**( ) can be used for ordinary, priority band, and high priority messages. However, unless precautions are taken, using **putbq( )** with a high priority message is likely to lead to an infinite loop of putting the message back on the queue, being rescheduled, pulling it off, and putting it back on.

This function is usually called when **bcanput**(9F) or **canput**(9F) determines that the message cannot be passed on to the next stream component. The flow control parameters are updated to reflect the change in the queue's status. If **QNOENB** is not set, the service routine is enabled.

RETURN VALUES

putbq( ) returns **1** on success and **0** on failure.

CONTEXT

putbq( ) can be called from user or interrupt context.

EXAMPLE

See the **bufcall**(9F) function page for an example of **putbq**( ).

SEE ALSO

bcanput(9F), **bufcall**(9F), **canput**(9F), **getq**(9F), **putq**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putctl – send a control message to a queue

SYNOPSIS

#include <sys/stream.h>

int putctl(queue_t *∗q*, **int** *type***);**

ARGUMENTS

q　　Queue to which the message is to be sent.

type　　Message type (must be control, not data type).

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putctl( ) tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. **putctl** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putctl( )** calls the **put**(9E) routine of the queue pointed to by *q* with the newly allocated and initialized messages.

RETURN VALUES

On success, **1** is returned. If *type* is a data type, or if a message block cannot be allocated, **0** is returned.

CONTEXT

putctl( ) can be called from user or interrupt context.

EXAMPLE

The **send_ctl** routine is used to pass control messages downstream. **M_BREAK** messages are handled with **putctl**( ) (line 11). **putctl1**(9F) (line 16) is used for **M_DELAY** messages, so that *parm* can be used to specify the length of the delay. In either case, if a message block cannot be allocated a variable recording the number of allocation failures is incremented (lines 12, 17). If an invalid message type is detected, **cmn_err**(9F) panics the system (line 21).

```
 1  void
 2  send_ctl(wrq, type, parm)
 3      queue_t *wrq;
 4      unchar type;
 5      unchar parm;
 6  {
 7          extern int num_alloc_fail;
 8
 9          switch (type) {
10          case M_BREAK:
11                  if (!putctl(wrq->q_next, M_BREAK))
12                          num_alloc_fail++;
13                  break;
14
15          case M_DELAY:
16                  if (!putctl1(wrq->q_next, M_DELAY, parm))
17                          num_alloc_fail++;
```

```
		break;

	default:
		cmn_err(CE_PANIC, "send_ctl: bad message type passed");
		break;
	}
}
```

SEE ALSO

put(9E), **cmn_err**(9F), **datamsg**(9F), **putctl1**(9F), **putnextctl**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putctl1 – send a control message with a one-byte parameter to a queue

SYNOPSIS

#include <sys/stream.h>

int putctl1(queue_t **q*, **int** *type*, **int** *p***);**

ARGUMENTS

q Queue to which the message is to be sent.

type Type of message.

p One-byte parameter.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putctl1( ), like **putctl**(9F), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *p* parameter can be used, for example, to specify how long the delay will be when an **M_DELAY** message is being sent. **putctl1**( ) fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a mesage block cannot be allocated. If successful, **putctl1**( ) calls the **put**(9E) routine of the queue pointed to by with the newly allocated and initialized message.

RETURN VALUES

On success, **1** is returned. **0** is returned if *type* is a data type, or if a message block cannot be allocated.

CONTEXT

putctl1( ) can be called from user or interrupt context.

EXAMPLE

See the **putctl**(9F) function page for an example of **putctl1( )**.

SEE ALSO

put(9E), **allocb**(9F), **datamsg**(9F), **putctl**(9F), **putnextctl1**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putnext – send a message to the next queue

SYNOPSIS

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

int putnext(queue_t **q*, **mblk_t** **mp***);**

ARGUMENTS

q Pointer to the queue from which the message *mp* will be sent.

mp Message to be passed.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putnext( ) is used to pass a message to the **put**(9E) routine of the next queue in the stream.

RETURN VALUES

None.

CONTEXT

putnext( ) can be called from user or interrupt context.

EXAMPLE

See **allocb**(9F) for an example of using **putnext**( ).

SEE ALSO

allocb(9F), **put**(9E)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putnextctl – send a control message to a queue

SYNOPSIS

#include <sys/stream.h>

int putnextctl(queue_t **q*, **int** *type***);**

ARGUMENTS

q	Queue to which the message is to be sent.
type	Message type (must be control, not data type).

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putnextctl( ) tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. **putnextctl**( ) fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putnextctl( )** calls the **put**(9E) routine of the queue pointed to by *q* with the newly allocated and initialized messages.

A call to **putnextctl(***q***,***type)* is an atomic equivalent of **putctl(***q*->*q_next***,***type)***.** The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing *q* through its **q_next** field and then invoking **putctl**(9F) proceeds without interference from other threads.

putnextctl( ) should always be used in preference to **putctl**(9F)**.**

RETURN VALUES

On success, **1** is returned. If *type* is a data type, or if a message block cannot be allocated, **0** is returned.

CONTEXT

putnextctl( ) can be called from user or interrupt context.

EXAMPLE

The **send_ctl** routine is used to pass control messages downstream. **M_BREAK** messages are handled with **putnextctl**( ) (line 8). **putnextctl1**(9F) (line 13) is used for **M_DELAY** messages, so that *parm* can be used to specify the length of the delay. In either case, if a message block cannot be allocated a variable recording the number of allocation failures is incremented (lines 9, 14). If an invalid message type is detected, **cmn_err**(9F) panics the system (line 18).

```
 1  void
 2  send_ctl(queue_t *wrq, u_char type, u_char parm)
 3  {
 4          extern int num_alloc_fail;
 5
 6          switch (type) {
 7          case M_BREAK:
 8                  if (!putnextctl(wrq, M_BREAK))
 9                          num_alloc_fail++;
10                  break;
11
12          case M_DELAY:
```

```
                    if (!putnextctl1(wrq, M_DELAY, parm))
                            num_alloc_fail++;
                    break;

            default:
                    cmn_err(CE_PANIC, "send_ctl: bad message type passed");
                    break;
            }
}
```

SEE ALSO **put**(9E), **cmn_err**(9F), **datamsg**(9F), **putctl**(9F), **putnextctl1**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putnextctl1 – send a control message with a one-byte parameter to a queue

SYNOPSIS

#include <sys/stream.h>

int putnextctl1(queue_t ∗*q***, int** *type***, int** *p***);**

ARGUMENTS

q Queue to which the message is to be sent.

type Type of message.

p One-byte parameter.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putnextctl1( ), like **putctl1**(9F), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *p* parameter can be used, for example, to specify how long the delay will be when an **M_DELAY** message is being sent. **putnextctl1**( ) fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putnextctl1**( ) calls the **put**(9E) routine of the queue pointed to by *q* with the newly allocated and initialized message.

A call to **putnextctl1(***q***,***type,* **p)** is an atomic equivalent of **putctl1(***q*->*q_next***,***type,* **p)**. The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing *q* through its **q_next** field and then invoking **putctl1**(9F) proceeds without interference from other threads.

putnextctl1( ) should always be used in preference to **putctl1**(9F)**.**

RETURN VALUES

On success, **1** is returned. **0** is returned if *type* is a data type, or if a message block cannot be allocated.

CONTEXT

putnextctl1( ) can be called from user or interrupt context.

EXAMPLE

See the **putnextctl**(9F) function page for an example of **putnextctl1**( )**.**

SEE ALSO

put(9E), **allocb**(9F), **datamsg**(9F), **putctl1**(9F), **putnextctl**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

putq – put a message on a queue

SYNOPSIS

#include <sys/stream.h>

int putq(queue_t **q*, **mblk_t** **bp***);**

ARGUMENTS

q Pointer to the queue to which the message is to be added.

bp Message to be put on the queue.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

putq( ) is used to put messages on a driver's queue after the module's put routine has finished processing the message. The message is placed after any other messages of the same priority, and flow control parameters are updated. If **QNOENB** is not set, the service routine is enabled. If no other processing is done, **putq** can be used as the module's put routine.

RETURN VALUES

putq( ) returns **1** on success and **0** on failure.

CONTEXT

putq( ) can be called from user or interrupt context.

EXAMPLE

See the **datamsg**(9F) function page for an example of **putq( )**.

SEE ALSO

datamsg(9F), **putbq**(9F), **qenable**(9F), **rmvq**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qbufcall – call a function when a buffer becomes available

SYNOPSIS

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

int qbufcall(queue_t **q,* **uint** *size,* **int** *pri,* **void (****func***)(long** *arg***),** **long** *arg* **);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

q Pointer to STREAMS queue structure.

size Number of bytes required for the buffer.

pri Priority of the **allocb**(9F) allocation request (not used).

func Function or driver routine to be called when a buffer becomes available.

arg Argument to the function to be called when a buffer becomes available.

DESCRIPTION

qbufcall serves as a **qtimeout**(9F) call of indeterminate length. When a buffer allocation request fails, **qbufcall( )** can be used to schedule the routine *func* to be called with the argument *arg* when a buffer becomes available. *func* may call **allocb( )** or it may do something else.

The **qbufcall( )** function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See **mt-streams**(9F) man page.) **qbufcall( )** schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to **qbufcall( )**. All outstanding bufcalls should be cancelled before the close of a driver or module returns.

qprocson(9F) must be called before calling either **qbufcall( )** or **qtimeout**(9F).

RETURN VALUES

If successful, **qbufcall( )** returns a **qbufcall** id that can be used in a call to **qunbufcall**(9F) to cancel the request. If the **qbufcall( )** scheduling fails, *func* is never called and **0** is returned.

CONTEXT

qbufcall( ) can be called from user or interrupt context.

WARNINGS

Even when *func* is called by **qbufcall( )**, **allocb**(9F) can fail if another module or driver had allocated the memory before *func* was able to call **allocb**(9F).

SEE ALSO

mt-streams(9F), **qprocson**(9F), **qtimeout**(9F), **qunbufcall**(9F), **quntimeout**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qenable – enable a queue

SYNOPSIS

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

void qenable(queue_t *∗q***);**

ARGUMENTS

q Pointer to the queue to be enabled.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

qenable( ) adds the queue pointed to by *q* to the list of queues whose service routines are ready to be called by the STREAMS scheduler.

CONTEXT

qenable( ) can be called from user or interrupt context.

EXAMPLE

See the **dupb**(9F) function page for an example of the **qenable( )**.

SEE ALSO

dupb(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qprocson, qprocsoff – enable, disable put and service routines

SYNOPSIS

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

void qprocson(queue_t **q***);**

void qprocsoff(queue_t **q***);**

ARGUMENTS

q Pointer to the RD side of a STREAMS queue pair.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

qprocson( ) enables the put and service routines of the driver or module whose read queue is pointed to by *q*. Threads cannot enter the module instance through the put and service routines while they are disabled.

qprocson( ) must be called by the open routine of a driver or module before returning, and after any initialization necessary for the proper functioning of the put and service routines.

qprocson( ) must be called before calling **qbufcall**(9F), **qtimeout**(9F), **qwait**(9F), or **qwait_sig**(9F),

qprocsoff( ) must be called by the close routine of a driver or module before returning, and before deallocating any resources necessary for the proper functioning of the put and service routines. It also removes the queue's service routines from the service queue, and blocks until any pending service processing completes.

The module or driver instance is guaranteed to be single-threaded before **qprocson( )** is called and after **qprocsoff( )** is called, except for threads executing asynchronous events such as interrupt handlers and callbacks, which must be handled separately.

CONTEXT

These routines can be called from user or interrupt context.

NOTES

The caller may not have the STREAM frozen during either of these calls.

SEE ALSO

close(9E), **open**(9E), **put**(9E), **srv**(9E), **qbufcall**(9F), **qtimeout**(9F), **qwait**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qreply – send a message on a stream in the reverse direction

SYNOPSIS

#include <sys/stream.h>

void qreply(queue_t **q*, **mblk_t** **mp***);**

ARGUMENTS

q Pointer to the queue.

mp Pointer to the message to be sent in the opposite direction.

INTERFACE LEVEL

Architecture independent level 1 (DDI/DKI).

DESCRIPTION

qreply( ) sends messages in the reverse direction of normal flow. That is, **qreply(***q***,***mp***)** is equivalent to **putnext(OTHERQ(***q***),***mp***).**

CONTEXT

qreply( ) can be called from user or interrupt context.

EXAMPLE

This example depicts the canonical flushing code for STREAMS drivers. Assume that the driver has service procedures (see **srv**(9E)), so that there may be messages on its queues. Its write-side put procedure (see **put**(9E)) handles **M_FLUSH** messages by first checking the **FLUSHW** bit in the first byte of the message, then the write queue is flushed (line 8) and the **FLUSHW** bit is turned off (line 9). If the **FLUSHR** bit is on, then the read queue is flushed (line 12) and the message is sent back up the read side of the stream with the **qreply**(9F) function (line 13). If the **FLUSHR** bit is off, then the message is freed (line 15). See the example for **flushq**(9F) for the canonical flushing code for modules.

```
 1 xxxwput(q, mp)
 2     queue_t *q;
 3     mblk_t *mp;
 4 {
 5          switch(mp->b_datap->db_type) {
 6          case M_FLUSH:
 7                  if (*mp->b_rptr & FLUSHW) {
 8                          flushq(q, FLUSHALL);
 9                          *mp->b_rptr &= ~FLUSHW;
10                  }
11                  if (*mp->b_rptr & FLUSHR) {
12                          flushq(RD(q), FLUSHALL);
13                          qreply(q, mp);
14                  } else {
15                          freemsg(mp);
16                  }
17                  break;
            . . .
18          }
19 }
```

SEE ALSO

put(9E), **srv**(9E), **flushq**(9F), **OTHERQ**(9F), **putnext**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qsize – find the number of messages on a queue

SYNOPSIS

#include <sys/stream.h>

int qsize(queue_t **q***);**

ARGUMENTS

q Queue to be evaluated.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

qsize( ) evaluates the queue *q* and returns the number of messages it contains.

RETURN VALUES

If there are no message on the queue, **qsize( )** returns **0**. Otherwise, it returns the integer representing the number of messages on the queue.

CONTEXT

qsize( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qtimeout – execute a function after a specified length of time

SYNOPSIS

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

int qtimeout(queue_t **q*, **void (****ftn*)(), **caddr_t** *arg*, **long** *ticks* **);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

q Pointer to STREAMS queue structure.

ftn Kernel function to invoke when the time increment expires.

arg Argument to the function.

ticks Number of clock ticks to wait before the function is called.

DESCRIPTION

The **qtimeout( )** function schedules the specified function *ftn* to be called after a specified time interval. *ftn* is called with *arg* as a parameter. Control is immediately returned to the caller. This is useful when an event is known to occur within a specific time frame, or when you want to wait for I / O processes when an interrupt is not available or might cause problems. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.

The **qtimeout( )** function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See **mt-streams**(9F) man page.) **qtimeout( )** schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to **qtimeout( )**. All outstanding timeouts should be cancelled before a driver closes or module returns.

qprocson(9F) must be called before calling **qtimeout( )**.

RETURN VALUES

Under normal conditions, an integer timeout identifier is returned.

The **qtimeout( )** function returns an identifier that may be passed to the **quntimeout**(9F) function to cancel a pending request. **Note:** No value is returned from the called function.

CONTEXT

qtimeout( ) can be called from user or interrupt context.

SEE ALSO

mt-streams(9F), **qbufcall**(9F), **qprocson**(9F), **qunbufcall**(9F), **quntimeout**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qunbufcall – cancel a pending qbufcall request

SYNOPSIS

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

void qunbufcall(queue_t **q*, **int** *id***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

q Pointer to STREAMS queue_t structure.

id Identifier returned from **qbufcall**(9F)

DESCRIPTION

qunbufcall cancels a pending **qbufcall( )** request. The argument *id* is a non-zero identifier of the request to be cancelled. *id* is returned from the **qbufcall( )** function used to issue the cancel request.

The **qunbufcall( )** function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See **mt-streams**(9F) man page.) **qunbufcall( )** returns when the bufcall has been cancelled or finished executing. The bufcall will be cancelled even if it is blocked at the perimeters associated with the queue. All outstanding bufcalls should be cancelled before the driver closes or module returns.

CONTEXT

qunbufcall( ) can be called from user or interrupt context.

SEE ALSO

mt-streams(9F), **qbufcall**(9F), **qtimeout**(9F), **quntimeout**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

quntimeout – cancel previous qtimeout function call

SYNOPSIS

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

int quntimeout(queue_t **q*, **int** *id* **);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

q Pointer to a STREAMS queue structure.

id Identification value generated by a previous **qtimeout**(9F) function call.

DESCRIPTION

quntimeout( ) cancels a pending **qtimeout**(9F) request. The **quntimeout( )** function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See **mt-streams**(9F) man page.) **quntimeout( )** returns when the timeout has been cancelled or finished executing. The timeout will be cancelled even if it is blocked at the perimeters associated with the queue. **quntimeout( )** should be executed for all outstanding timeouts before a driver or module close returns.

RETURN VALUES

quntimeout( ) returns **-1** if the *id* is not found. Otherwise, **quntimeout( )** returns a zero or positive value.

CONTEXT

quntimeout( ) can be called from user or interrupt context.

SEE ALSO

mt-streams(9F), **qbufcall**(9F), **qtimeout**(9F), **qunbufcall**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

qwait, qwait_sig – STREAMS wait routines

SYNOPSIS

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

void qwait(queue_t **q***);**

int qwait_sig(queue_t **q***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

qp Pointer to the queue that is being opened or closed.

DESCRIPTION

qwait( ) and **qwait_sig( )** are used to wait for a message to arrive to the **put**(9E) or **srv**(9E) procedures. They can be used in the **open**(9E) and **close**(9E) procedures in a STREAMS driver or module. **qwait( )** and **qwait_sig( )** atomically exit the inner and outer perimeters associated with the queue, and wait for a thread to leave modules **put**(9E) or **srv**(9E) procedures. Upon return they re-enter the inner and outer perimeters.

qprocson(9F) must be called before calling **qwait( )** or **qwait_sig( )**.

qwait( ) is not interrupted by a signal, whereas **qwait_sig( )** is interrupted by a signal. **qwait_sig( )** normally returns non-zero, and returns zero when the waiting was interrupted by a signal.

qwait( ) and **qwait_sig( )** are similar to **cv_wait( )** and **cv_wait_sig( )** (see **condvar**(9F)), except that the mutex is replaced by the inner and outer perimeters and the signalling is implicit when a thread leaves the inner perimeter.

RETURN VALUES

0 For **qwait_sig( )**, indicates that the condition was not necessarily signaled and the function returned because a signal was pending.

CONTEXT

These functions can only be called from an **open**(9E) or **close**(9E) routine.

EXAMPLES

The open routine sends down a **T_INFO_REQ** message and waits for the **T_INFO_ACK**. The arrival of the **T_INFO_ACK** is recorded by resetting a flag in the unit structure (**WAIT_INFO_ACK**).

The example assumes that the module is **D_MTQPAIR** or **D_MTPERMOD**.

```
xxopen(qp, . . .)
        queue_t *qp;
{
        struct xxdata *xx;

        /* Allocate xxdata structure */
        qprocson(qp);
        /* Format T_INFO_ACK in mp */
        putnext(qp, mp);
        xx->xx_flags |= WAIT_INFO_ACK;
```

```
                while (xx->xx_flags & WAIT_INFO_ACK)
                        qwait(qp);
                return (0);
        }

        xxrput(qp, mp)
                queue_t *qp;
                mblk_t *mp;
        {
                struct xxdata *xx = (struct xxdata *)q->q_ptr;

                ...

                case T_INFO_ACK:
                        if (xx->xx_flags & WAIT_INFO_ACK) {
                                /* Record information from info ack */
                                xx->xx_flags &= ˜WAIT_INFO_ACK;
                                freemsg(mp);
                                return;
                        }

                ...
        }
```

SEE ALSO **close**(9E), **open**(9E), **put**(9E), **srv**(9E) **condvar**(9F), **mt-streams**(9F), **qprocson**(9F)

STREAMS Programmer's Guide

Writing Device Drivers

NAME

qwriter – asynchronous STREAMS perimeter upgrade

SYNOPSIS

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

void qwriter(queue_t ∗*qp,* **mblk_t** ∗*mp,* **void (**∗*func***)( ), int** *perimeter***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

qp Pointer to the queue.

mp Pointer to a message that will be passed in to the callback function.

func A function that will be called when exclusive (writer) access has been acquired at the specified perimeter.

perimeter Either **PERIM_INNER** or **PERIM_OUTER**.

DESCRIPTION

qwriter( ) is used to upgrade the access at either the inner or the outer perimeter from shared to exclusive (see **mt-streams**(9F) man page), and call the specified callback function when the upgrade has succeeded. The callback function is called as:

(∗**func)(queue_t** ∗*qp***, mblk_t** ∗*mp***);**

qwriter( ) will acquire exclusive access immediately if possible, in which case the specified callback function will be executed before **qwriter( )** returns. If this is not possible, **qwriter( )** will defer the upgrade until later and return before the callback function has been executed. Modules should not assume that the callback function has been executed when **qwriter( )** returns. One way to avoid dependencies on the execution of the callback function is to immediately return after calling **qwriter( )** and let the callback function finish the processing of the message.

When **qwriter( )** defers calling the callback function, the STREAMS framework will prevent other messages from entering the inner perimeter associated with the queue until the upgrade has completed and the callback function has finished executing.

CONTEXT

qwriter( ) can only be called from an **put**(9E) or **srv**(9E) routine, or from a **qwriter**( )**,** **qtimeout**(9F), or **qbufcall**(9F) callback function.

SEE ALSO

put(9E), **srv**(9E), **mt-streams**(9F), **qbufcall**(9F), **qtimeout**(9F)

STREAMS Programmer's Guide
Writing Device Drivers

NAME

rmalloc – allocate space from a resource map

SYNOPSIS

#include <sys/map.h>
#include <sys/ddi.h>

unsigned long rmalloc(struct map **mp*, **size_t** *size***);**

ARGUMENTS

mp Resource map from where the resource is drawn.

size Number of units of the resource.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

rmalloc( ) is used by a driver to allocate space from a previously defined and initialized resource map. The map itself is allocated by calling the function **rmallocmap**(9F). **rmalloc( )** is one of five functions used for resource map management. The other functions include:

rmalloc_wait(9F)	Allocate space from a resource map, wait if necessary.
rmfree(9F)	Return previously allocated space to a map.
rmallocmap(9F)	Allocate a resource map initialize it.
rmfreemap(9F)	Deallocate a resource map.

rmalloc( ) allocates space from a resource map in terms of arbitrary units. The system maintains the resource map by size and index, computed in units appropriate for the resource. For example, units may be byte addresses, pages of memory, or blocks. The normal return value is an **unsigned long** set to the value of the index where sufficient free space in the resource was found.

RETURN VALUES

Under normal conditions, **rmalloc( )** returns the base index of the allocated space. Otherwise, **rmalloc( )** returns a **0** if all resource map entries are already allocated.

CONTEXT

rmalloc( ) can be called from user or interrupt context.

EXAMPLE

The following example is a simple memory map, but it illustrates the principles of map management. A driver allocates and initializes the map by calling both the **rmallocmap**(9F) and **rmfree**(9F) functions. **rmallocmap**(9F) is called to establish the number of slots or entries in the map, and **rmfree**(9F) to initialize the resource area the map is to manage. The following example is a fragment from a hypothetical **start** routine and illustrates the following procedures:

Panics the system if the required amount of memory can not be allocated (lines 11–15).

Uses **rmallocmap**(9F) to configure the total number of entries in the map, and **rmfree**(9F) to initialize the total resource area.

```
1  #define XX_MAPSIZE        12
2  #define XX_BUFSIZE  2560
3  static struct map *xx_mp;       /* Private buffer space map */
   . . .
4  xxstart()
5      /*
6       *  Allocate private buffer.  If insufficient memory,
7       *  display message and halt system.
8       */
9  {
10     register caddr_t bp;
   . . .
11     if ((bp = kmem_alloc(XX_BUFSIZE, KM_NOSLEEP) == 0)  {
12
13         cmn_err(CE_PANIC, "xxstart: kmem_alloc failed before %d buffer"
14                 "allocation", XX_BUFSIZE);
15     }
16
17     /*
18      * Initialize the resource map with number
19      * of slots in map.
20      */
21     xx_mp = rmallocmap(XX_MAPSIZE);
22
24     /*
25      * Initialize space management map with total
26      * buffer area it is to manage.
27      */
28     rmfree(xx_mp, XX_BUFSIZE, bp);
       . . .
```

The **rmalloc( )** function is then used by the driver's **read** or **write** routine to allocate buffers for specific data transfers. The **uiomove**(9F) function is used to move the data between user space and local driver memory. The device then moves data between itself and local driver memory through DMA.

The next example illustrates the following procedures:

> The size of the I/O request is calculated and stored in the *size* variable (line 10).
>
> Buffers are allocated through the **rmalloc**(9F) function using the *size* value (line 15). If the allocation fails the system will panic.
>
> The **uiomove**(9F) function is used to move data to the allocated buffer (line 23).
>
> If the address passed to **uiomove**(9F) is invalid, **rmfree**(9F) is called to release the previously allocated buffer, and an **EFAULT** error is returned.

```
#define XX_BUFSIZE  2560
#define XX_MAXSIZE  (XX_BUFSIZE / 4)

static struct map *xx_mp;        /* Private buffer space map */
...
xxread(dev_t dev, uio_t *uiop, cred_t *credp)
{

register caddr_t addr;
register int     size;
    size = min(COUNT, XX_MAXSIZE);  /* Break large I/O request */
                                    /* into small ones */
     /*
      * Get buffer.
      */
    if ((addr = (caddr_t)rmalloc(xx_mp, size)) == 0)
       cmn_err(CE_PANIC, "read: rmalloc failed allocation of size %d",
              size);

     /*
      * Move data to buffer.  If invalid address is found,
      * return buffer to map and return error code.
      */
    if (uiomove(addr, size, UIO_READ, uiop) == –1)  {
       rmfree(xx_mp, size, addr);
       return(EFAULT);
    }
}
```

SEE ALSO

kmem_alloc(9F), **rmalloc_wait**(9F), **rmallocmap**(9F), **rmfree**(9F), **rmfreemap**(9F), **uiomove**(9F)

Writing Device Drivers

NAME

rmalloc_wait – allocate space from a resource map, wait if necessary

SYNOPSIS

#include <sys/map.h>
#include <sys/ddi.h>

unsigned long rmalloc_wait(struct map ∗*mp*, **size_t** *size***);**

ARGUMENTS

mp Pointer to the resource map from which space is to be allocated.

size Number of units of space to allocate.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

rmalloc_wait( ) requests an allocation of space from a resource map. **rmalloc_wait( )** is similar to the **rmalloc**(9F) function with the exception that it will wait for space to become available if necessary.

RETURN VALUES

rmalloc_wait( ) returns the base of the allocated space.

CONTEXT

This functions can be called from user or interrupt context. However in most cases **rmalloc_wait( )** should be called from user context only.

SEE ALSO

rmalloc(9F), **rmallocmap**(9F), **rmfree**(9F), **rmfreemap**(9F)

Writing Device Drivers

NAME

rmallocmap, rmfreemap – allocate and free (respectively) resource maps

SYNOPSIS

#include <sys/map.h>
#include <sys/ddi.h>

struct map ∗rmallocmap(unsigned long *mapsize***);**

void rmfreemap(struct map ∗*mp***);**

ARGUMENTS

mapsize Number of entries for the map.

mp A pointer to the map structure to be deallocated.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

rmallocmap( ) dynamically allocates a resource map structure. The argument *mapsize* defines the total number of entries in the map. In particular it is the total number allocations that can be outstanding at any one time.

rmallocmap( ) initializes the map but does not associate it with the actual resource. In order to associate the map with the actual resource a call to **rmfree**(9F) is used to make the entirety of the actual resource available for allocation starting from the first index into the resource. Typically the call to **rmallocmap( )** is followed by a call to **rmfree**(9F), passing the address of the map returned from **rmallocmap( )**, the total size of the resource, and the first index into actual resource.

The resource map allocated by **rmallocmap( )** can be used to describe an arbitrary resource in whatever allocation units are appropriate such blocks, pages, or data structures. This resource can then be managed by the system by subsequent calls to **rmalloc**(9F), **rmalloc_wait**(9F), and **rmfree**(9F).

rmfreemap( ) deallocates a resource map structure previously allocated by **rmallocmap( )**. The argument *mp* is a pointer to the map structure to be deallocated.

RETURN VALUES

Upon successful completion, **rmallocmap( )** returns a pointer to the newly allocated map structure. Upon failure, **rmallocmap( )** returns a NULL pointer.

CONTEXT

rmallocmap( ) can be called from user or interrupt context.

SEE ALSO

rmalloc(9F), **rmalloc_wait**(9F), **rmfree**(9F)

Writing Device Drivers

NAME

rmfree – free space back into a resource map

SYNOPSIS

#include <sys/map.h>
#include <sys/ddi.h>

void rmfree(struct map *∗mp***, size_t** *size***, ulong_t** *index***);**

ARGUMENTS

mp Pointer to the map structure.

size Number of units being freed.

index Index of the first unit of the allocated resource.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

rmfree( ) releases space back into a resource map. It is the opposite of **rmalloc**(9F), which allocates space that is controlled by a resource map structure.

Drivers may define resource maps for resource allocation, in terms of arbitrary units, using the **rmallocmap**(9F), function. The system maintains the resource map structure by size and index, computed in units appropriate for the resource. For example, units may be byte addresses, pages of memory, or blocks. **rmfree( )** frees up unallocated space for re-use.

CONTEXT

rmfree( ) can be called from user or interrupt context.

SEE ALSO

rmalloc(9F), **rmalloc_wait**(9F), **rmallocmap**(9F), **rmfreemap**(9F)

Writing Device Drivers

NAME

rmvb – remove a message block from a message

SYNOPSIS

#include <sys/stream.h>
mblk_t *rmvb(mblk_t ***mp*, mblk_t ***bp***);**

ARGUMENTS

mp Message from which a block is to be removed. **mblk_t** is an instance of the **msgb**(9S) structure.

bp Message block to be removed.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

rmvb( ) removes a message block (*bp*) from a message (*mp*), and returns a pointer to the altered message. The message block is not freed, merely removed from the message. It is the module or driver's responsibility to free the message block.

RETURN VALUES

If successful, a pointer to the message (minus the removed block) is returned. The pointer is **NULL** if *bp* was the only block of the message before **rmvb( )** was called. If the designated message block (*bp*) does not exist, **-1** is returned.

CONTEXT

rmvb( ) can be called from user or interrupt context.

EXAMPLE

This routine removes all zero-length **M_DATA** message blocks from the given message. For each message block in the message, save the next message block (line 10). If the current message block is of type **M_DATA** and has no data in its buffer (line 11), then remove it from the message (line 12) and free it (line 13). In either case, continue with the next message block in the message (line 16).

```
 1  void
 2  xxclean(mp)
 3      mblk_t *mp;
 4  {
 5          mblk_t *tmp;
 6          mblk_t *nmp;
 7
 8          tmp = mp;
 9          while (tmp) {
10                  nmp = tmp->b_cont;
11                  if ((tmp->b_datap->db_type == M_DATA) &&
                        (tmp->b_rptr == tmp->b_wptr)) {
12                          (void) rmvb(mp, tmp);
13                          freeb(tmp);
14                  }
15                  tmp = nmp;
16          }
17  }
```

SEE ALSO

freeb(9F), **msgb**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

rmvq – remove a message from a queue

SYNOPSIS

#include <sys/stream.h>

void rmvq(queue_t **q*, **mblk_t** **mp***);**

ARGUMENTS

q Queue containing the message to be removed.

mp Message to remove.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

rmvq( ) removes a message from a queue. A message can be removed from anywhere on a queue. To prevent modules and drivers from having to deal with the internals of message linkage on a queue, either **rmvq**( ) or **getq**(9F) should be used to remove a message from a queue.

CONTEXT

rmvq( ) can be called from user or interrupt context.

EXAMPLE

This code fragment illustrates how one may flush one type of message from a queue. In this case, only **M_PROTO T_DATA_IND** messages are flushed. For each message on the queue, if it is an **M_PROTO** message (line 8) of type **T_DATA_IND** (line 10), save a pointer to the next message (line 11), remove the **T_DATA_IND** message (line 12) and free it (line 13). Continue with the next message in the list (line 19).

```
 1  mblk_t *mp, *nmp;
 2  queue_t *q;
 3  union T_primitives *tp;
 4
 5          freezestr(q);
 6          mp = q->q_first;
 7          while (mp) {
 8                  if (mp->b_datap->db_type == M_PROTO) {
 9                          tp = (union T_primitives *)mp->b_rptr;
10                          if (tp->type == T_DATA_IND) {
11                                  nmp = mp->b_next;
12                                  rmvq(q, mp);
13                                  freemsg(mp);
14                                  mp = nmp;
15                          } else {
16                                  mp = mp->b_next;
17                          }
18                  } else {
19                          mp = mp->b_next;
20                  }
21          }
22          unfreezestr(q);
```

SEE ALSO

freemsg(9F), **freezestr**(9F), **getq**(9F), **insq**(9F), **unfreezestr**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

WARNINGS

Make sure that the message *mp* is linked onto *q* to avoid a possible system panic.

NOTES

The stream must be frozen using **freezestr**(9F) before calling **rmvq( ).**

NAME

rwlock, rw_init, rw_destroy, rw_enter, rw_exit, rw_tryenter, rw_downgrade, rw_tryupgrade, rw_read_locked – readers / writer lock functions

SYNOPSIS

#include <sys/ksynch.h>

void rw_init(krwlock_t **rwlp*, **char** **name*, **krw_type_t** *type*, **void** **arg***);**

void rw_destroy(krwlock_t **rwlp***);**

void rw_enter(krwlock_t **rwlp*, **krw_t** *enter_type***);**

void rw_exit(krwlock_t **rwlp***);**

int rw_tryenter(krwlock_t **rwlp*, **krw_t** *enter_type***);**

void rw_downgrade(krwlock_t **rwlp***);**

int rw_tryupgrade(krwlock_t **rwlp***);**

int rw_read_locked(krwlock_t **rwlp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

rwlp	Pointer to a **krwlock_t** readers / writer lock.
name	Character string describing lock for statistics and debugging.
type	Type of readers / writer lock.
arg	Type-specific argument for initialization function.
enter_type	Indication of whether the lock is to be acquired non-exclusively or exclusively **RW_READER** or **RW_WRITER**.

DESCRIPTION

A multiple-readers, single-writer lock is represented by the **krwlock_t** data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers / writer lock.

Readers / writer locks can be more than twice as expensive as a mutex lock, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

rw_init initializes a readers / writer lock. It is an error to initialize a lock more than once. The *type* argument should be set to **RW_DRIVER**. The type-specific argument, *arg*, should be the ddi_iblock_cookie returned from **ddi_add_intr**(9F) if the lock is used by the interrupt handler. If the lock is not used by any interrupt handler, the argument should be NULL.

If the call to **rw_init** is compiled with **_LOCKTEST** or **_MPSTATS** defined, statistics will be kept for the lock. This may have a performance penalty.

rw_destroy releases any storage that might have been allocated by **rw_init**. It should be called before deallocating the storage containing the lock.

rw_enter acquires the lock, and blocks if necessary. If *enter_type* is **RW_READER**, the caller blocks if there is a writer or a thread attempting to enter for writing. If *enter_type* is **RW_WRITER**, the caller blocks if any thread holds the lock.

rw_exit releases the lock and may wake up one or more threads waiting on the lock.

rw_tryenter attempts to enter the lock, like **rw_enter**, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with **RW_WRITER**), may call **rw_downgrade** to convert to holding the lock non-exclusively (as if entered with **RW_READER**). Other waiting readers will be unblocked unless there is a waiting writer.

rw_tryupgrade can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

rw_read_locked returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if **rw_read_locked** is called for a lock that isn't held by the caller.

RETURN VALUES

0	**rw_tryenter** could not obtain the lock without blocking.
0	**rw_tryupgrade** was unable to perform the upgrade because of other threads holding or waiting to hold the lock.
0	**rw_read_locked** returns 0 if the lock is held by the caller for write.
non-zero	from **rw_read_locked** if the lock is held by the caller for read. non-zero successful return from **rw_tryenter** or **rw_tryupgrade**.

CONTEXT

These functions can be called from user or interrupt context, except for **rw_init** and **rw_destroy**, which can be called from user context only.

SEE ALSO

condvar(9F), **ddi_add_intr**(9F), **mutex**(9F), **semaphore**(9F)

Writing Device Drivers

NAME

scsi_abort – abort a SCSI command

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_abort(struct scsi_address **ap*, **struct scsi_pkt** **pkt*);

ARGUMENTS

ap Pointer to a **scsi_address** structure.

pkt Pointer to a **scsi_pkt**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_abort( ) terminates a command that has been transported to the host adapter driver. A NULL *pkt* causes all outstanding packets to be aborted. On a successful abort, the *pkt_reason* is set to **CMD_ABORTED** and *pkt_statistics* is updated.

RETURN VALUES

scsi_abort( ) returns:

1 on success.

0 on failure.

CONTEXT

scsi_abort( ) can be called from user or interrupt context.

EXAMPLE

```
if (scsi_abort(&devp->sd_address, pkt) == 0) {
        (void) scsi_reset(&devp->sd_address, RESET_ALL);
}
```

SEE ALSO

scsi_reset(9F), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_alloc_consistent_buf – allocate an I/O buffer for SCSI DMA

SYNOPSIS

#include <sys/scsi/scsi.h>

struct buf *__scsi_alloc_consistent_buf(struct scsi_address__ **ap*, **struct buf** **bp*, **int** *datalen*, **ulong** *bflags*, **int (***waitfunc* **)(caddr_t), caddr_t** *arg***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ap Pointer to the **scsi_address**(9S) structure.

bp Pointer to the **buf**(9S) structure.

datalen Number of bytes for the data buffer.

bflags Flags setting for the allocated buffer header.

waitfunc Pointer to either **NULL_FUNC** or **SLEEP_FUNC**.

arg *waitfunc* function argument, must be **NULL**.

DESCRIPTION

scsi_alloc_consistent_buf( ) allocates a buffer header and the associated data buffer for direct memory access (DMA) transfer. This buffer is allocated from the *iobp* space, which is considered consistent memory. For more details, see **ddi_iopb_alloc**(9F) and **ddi_dma_sync**(9F).

For buffers allocated via **scsi_alloc_consistent_buf( )**, and marked with the **PKT_CONSISTENT** flag via **scsi_init_pkt**(9F), the HBA driver must ensure that the data transfer for the command is correctly synchronized before the target driver's command completion callback is performed.

If *bp* is **NULL**, a new buffer header will be allocated using **getrbuf**(9F). In addition, if *datalen* is non-zero, a new buffer will be allocated using **ddi_iopb_alloc**(9F).

waitfunc indicates what the allocator routines should do when direct memory access (DMA) resources are not available; the valid values are:

NULL_FUNC Do not wait for resources. Return a **NULL** pointer.

SLEEP_FUNC Wait indefinitely for resources.

RETURN VALUES

scsi_alloc_consistent_buf( ) returns a pointer to a **buf**(9S) structure on success. It returns NULL if resources are not available and *waitfunc* was not **SLEEP_FUNC**.

CONTEXT

If *waitfunc* is **SLEEP_FUNC**, then this routine may be called only from user-level code. Otherwise, it may be called from either user or interrupt level. The *waitfunc* function may not block or call routines that block.

EXAMPLE

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
    SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
rqpkt = scsi_init_pkt(&devp->sd_address,
    NULL, bp, CDB_GROUP0, 1, 0,
    PKT_CONSISTENT, SLEEP_FUNC, NULL);
```

SEE ALSO

ddi_dma_sync(9F), **ddi_iopb_alloc**(9F), **getrbuf**(9F), **scsi_init_pkt**(9F), **scsi_destroy_pkt**(9F), **scsi_free_consistent_buf**(9F), **buf**(9S)

Writing Device Drivers

NAME

scsi_cname, scsi_dname, scsi_mname, scsi_rname, scsi_sname – decode a SCSI name

SYNOPSIS

#include <sys/scsi/scsi.h>

char *scsi_cname(u_char *cmd***, char** ***cmdvec***);**

char *scsi_dname(int *dtype***);**

char *scsi_mname(u_char *msg***);**

char *scsi_rname(u_char *reason***);**

char *scsi_sname(u_char *sense_key***);**

ARGUMENTS

cmd	A SCSI command value.
cmdvec	Pointer to an array of command strings.
dtype	Device type.
msg	A message value.
reason	A packet reason value.
sense_key	A SCSI sense key value.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_cname( ) decodes SCSI commands. *cmdvec* is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.

scsi_dname( ) decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

scsi_mname( ) decodes SCSI messages.

scsi_rname( ) decodes packet completion reasons.

scsi_sname( ) decodes SCSI sense keys.

RETURN VALUES

These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLE

scsi_cname( ) decodes SCSI commands as follows:

```
static char *st_cmds[] = {
        "\000test unit ready",
        "\001rewind",
        "\003request sense",
        "\010read",
        "\012write",
        "\020write file mark",
        "\021space",
        "\022inquiry",
        "\025mode select",
        "\031erase tape",
        "\032mode sense",
        "\033load tape",
        NULL
};
..
cmn_err(CE_CONT, "st: cmd=%s", scsi_cname(cmd, st_cmds));
..
```

SEE ALSO

Writing Device Drivers

NAME

scsi_destroy_pkt – free an allocated SCSI packet and its DMA resource

SYNOPSIS

#include <sys/scsi/scsi.h>

void scsi_destroy_pkt(struct scsi_pkt **pktp*);

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

pktp Pointer to a **scsi_pkt**(9S) structure.

DESCRIPTION

scsi_destroy_pkt( ) releases all necessary resources, typically at the end of an I/O transfer. The data is synchronized to memory, then the DMA resources are deallocated and *pktp* is freed.

CONTEXT

scsi_destroy_pkt( ) may be called from user or interrupt context.

EXAMPLE

```
scsi_destroy_pkt(un->un_rqs);
```

SEE ALSO

scsi_init_pkt(9F), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_dmaget, scsi_dmafree – SCSI dma utility routines

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_dmaget(struct scsi_pkt **pkt*, **opaque_t***dmatoken*,
int (**callback***)(void));**

void scsi_dmafree(struct scsi_pkt **pkt***);**

ARGUMENTS

pkt A pointer to a **scsi_pkt**(9S) structure.

dmatoken Pointer to an implementation dependent object

callback Pointer to a callback function, or **NULL_FUNC** or **SLEEP_FUNC.**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_dmaget( ) allocates DMA resources for an already allocated SCSI packet. *pkt* is a pointer to the previously allocated SCSI packet (see **scsi_pktalloc**(9F)).

dmatoken is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The *dmatoken* must be a pointer to a **buf**(9S) structure. If *dmatoken* is **NULL**, no resources are allocated.

callback indicates what **scsi_dmaget( )** should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a NULL pointer.

SLEEP_FUNC Wait indefinitely for resources.

Other Values *callback* points to a function which is called when resources may have become available. *callback* **must** return either **0** (indicating that it attempted to allocate resouces but failed to do so again), in which case it is put back on a list to be called again later, or **1** indicating either success in allocating resources or indicating that it no longer cares for a retry.

scsi_dmafree( ) frees the DMA resources associated with the SCSI packet. The packet itself remains allocated.

RETURN VALUES

scsi_dmaget( ) returns a pointer to a **scsi_pkt** on success. It returns **NULL** if resources are not available.

CONTEXT

If *callback* is **SLEEP_FUNC**, then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block.

scsi_dmafree( ) can be called from user or interrupt context.

SEE ALSO

scsi_pktalloc(9F), **scsi_pktfree**(9F), **scsi_resalloc**(9F), **scsi_resfree**(9F), **buf**(9S), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_errmsg – display a SCSI request sense message

SYNOPSIS

#include <sys/scsi/scsi.h>

void scsi_errmsg(struct scsi_device **devp*, **struct scsi_pkt** **pktp*, **char** **drv_name*,
int *severity*, **int** *blkno*, **int** *err_blkno*, **struct scsi_key_strings** **cmdlist*,
struct scsi_extended_sense **sensep*);

ARGUMENTS

devp	Pointer to the **scsi_device**(9S) structure.
pktp	Pointer to a **scsi_pkt**(9S) structure.
drv_name	String used by **scsi_log**(9F).
severity	Error severity level, maps to severity strings below.
blkno	Requested block number.
err_blkno	Error block number.
cmdlist	An array of SCSI command description strings.
sensep	A pointer to a **scsi_extended_sense**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_errmsg( ) interprets the request sense information in the *sensep* pointer and generates a standard message that is displayed using **scsi_log**(9F). The first line of the message is always a CE_WARN, with the continuation lines being CE_CONT. *sensep* may be NULL in which case no sense key or vendor information is displayed.

The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The **scsi_device**(9S) structure denoted by *devp* supplies the identification of the device that requested the display. *severity* selects which string is used in the "Error Level:" reporting, according to the table below:

Severity Value:	**String:**
SCSI_ERR_ALL	**All**
SCSI_ERR_UNKNOWN	**Unknown**
SCSI_ERR_INFO	**Information**
SCSI_ERR_RECOVERED	**Recovered**
SCSI_ERR_RETRYABLE	**Retryable**
SCSI_ERR_FATAL	**Fatal**

blkno is the block number of the original request that generated the error. *err_blkno* is the block number where the error occurred. *cmdlist* is a mapping table for translating the SCSI command code in *pktp* to the actual command string.

The *cmdlist* is described in the structure below:

```
struct scsi_key_strings {
        int key;
        char *message;
};
```

For a basic SCSI disk the following list is appropriate:

```
static struct scsi_key_strings sd_cmds[] = {
        0x00, "test unit ready",
        0x01, "rezero",
        0x03, "request sense",
        0x04, "format",
        0x07, "reassign",
        0x08, "read",
        0x0a, "write",
        0x0b, "seek",
        0x12, "inquiry",
        0x15, "mode select",
        0x16, "reserve",
        0x17, "release",
        0x18, "copy",
        0x1a, "mode sense",
        0x1b, "start/stop",
        0x1e, "door lock",
        0x28, "read(10)",
        0x2a, "write(10)",
        0x2f, "verify",
        0x37, "read defect data",
        −1, NULL
};
```

CONTEXT

scsi_errmsg( ) may be called from user or interrupt context.

EXAMPLE

```
    scsi_errmsg(devp, pkt, "sd", SCSI_ERR_INFO, bp->b_blkno,
        err_blkno, sd_cmds, rqsense);
```

Generates:

```
WARNING: /sbus@1,f8000000/esp@0,800000/sd@1,0 (sd1):
        Error for command 'read'        Error Level: Informational
        Requested Block 23936, Error Block: 23936
        Sense Key: Unit Attention
        Vendor 'QUANTUM': ASC = 0x29 (reset), ASCQ = 0x0, FRU = 0x0
```

SEE ALSO

cmn_err(9F), **scsi_log**(9F), **scsi_device**(9S), **scsi_extended_sense**(9S), **scsi_pkt**(9S)
Writing Device Drivers

NAME

scsi_free_consistent_buf – free a previously allocated SCSI DMA I/O buffer

SYNOPSIS

#include <sys/scsi/scsi.h>

void scsi_free_consistent_buf(struct buf **bp*);**

ARGUMENTS

bp Pointer to the **buf**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_free_consistent_buf() frees a buffer header and consistent data buffer that was previously allocated using **scsi_alloc_consistent_buf**(9F).

CONTEXT

scsi_free_consistent_buf() may be called from either the user or the interrupt levels.

SEE ALSO

freerbuf(9F), **scsi_alloc_consistent_buf**(9F), **buf**(9S)

Writing Device Drivers

WARNING

scsi_free_consistent_buf() will call **freerbuf**(9F) to free the **buf**(9S) that was allocated before or during the call to **scsi_alloc_consistent_buf**(9F).

NAME

scsi_hba_attach, scsi_hba_detach – SCSI HBA attach and detach routines

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_attach(dev_info_t **dip*, **ddi_dma_lim_t** **hba_lim*,
    scsi_hba_tran_t **hba_tran*, **int** *hba_flags*, **void** **hba_options*);

int scsi_hba_detach(dev_info_t **dip*);

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

dip	A pointer to the **dev_info_t** structure, referring to the instance of the HBA device.
hba_lim	A pointer to a **ddi_dma_lim**(9S) structure.
hba_tran	A pointer to a **scsi_hba_tran**(9S) structure
hba_flags	flag modifiers. The only defined flag value is **SCSI_HBA_TRAN_CLONE**.
hba_options	optional features provided by the HBA driver for future extensions; must be NULL.

DESCRIPTION

scsi_hba_attach( )

scsi_hba_attach( ) registers the DMA limits *hba_lim* and the transport vectors *hba_tran* of each instance of the HBA device defined by *dip*. The HBA driver can pass different DMA limits and transport vectors for each instance of the device, as necessary, to support any constraints imposed by the HBA itself.

scsi_hba_attach( ) uses the **dev_bus_ops** field in the **dev_ops** structure. The HBA driver should initialize this field to **NULL** before calling **scsi_hba_attach( ).**

If **SCSI_HBA_TRAN_CLONE** is requested in *hba_flags*, the *hba_tran* structure will be cloned once for each target attached to the HBA. The cloning of the structure will occur before the **tran_tgt_init**(9E) entry point is called to initialize a target. At all subsequent HBA entry points, including **tran_tgt_init**(9E), the **scsi_hba_tran_t** structure passed as an argument or found in a **scsi_address** structure will be the 'cloned' **scsi_hba_tran_t** structure, thus allowing the HBA to use the **tran_tgt_private** field in the **scsi_hba_tran_t** structure to point to per-target data. The HBA must take care to free only the same **scsi_hba_tran_t** structure it allocated when detaching; all 'cloned' **scsi_hba_tran_t** structures allocated by the system will be freed by the system.

scsi_hba_attach( ) attaches a number of integer-valued properties to *dip*, via **ddi_prop_create**(9F), unless properties of the same name are already attached to the node. An HBA driver should retrieve these configuration parameters via **ddi_prop_op**(9F), and respect any settings for features provided the HBA.

scsi-options   optional SCSI configuration bits

  SCSI_OPTIONS_DR   if not set, the HBA should not grant Disconnect privileges to target devices.

SCSI_OPTIONS_LINK	if not set, the HBA should not enable Linked Commands.
SCSI_OPTIONS_TAG	if not set, the HBA should not operate in Command Tagged Queueing mode.
SCSI_OPTIONS_FAST	if not set, the HBA should not operate the bus in FAST SCSI mode.
SCSI_OPTIONS_WIDE	if not set, the HBA should not operate the bus in WIDE SCSI mode.

scsi-reset-delay SCSI bus or device reset recovery time, in milliseconds.

scsi_hba_detach( )

scsi_hba_detach( ) removes the DMA limits structure and the transport vector for the given instance of an HBA driver.

RETURN VALUES

scsi_hba_attach( ) and **scsi_hba_detach( )** return **DDI_SUCCESS** if the function call succeeds, and returns **DDI_FAILURE** on failure.

CONTEXT

scsi_hba_attach( ) and **scsi_hba_detach( )** should be called from **attach**(9E) or **detach**(9E), respectively.

NOTES

It is the HBAs responsibility to ensure that no more transport requests will be taken on behalf of any SCSI target device driver after **scsi_hba_detach( )** is called.

SEE ALSO

attach(9E), **detach**(9E), **tran_tgt_init**(9E), **ddi_prop_create**(9F), **scsi_address**(9S), **scsi_hba_tran**(9S)

Writing Device Drivers

NAME

scsi_hba_init, scsi_hba_fini – SCSI Host Bus Adapter system initialization and completion routines

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_init(struct modlinkage **modlp***);**

void scsi_hba_fini(struct modlinkage **modlp***);**

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

modlp Pointer to the Host Bus Adapters module linkage structure.

DESCRIPTION

scsi_hba_init( )

scsi_hba_init( ) is the system-provided initialization routine for SCSI HBA drivers. The **scsi_hba_init( )** function registers the HBA in the system and allows the driver to accept configuration requests on behalf of SCSI target drivers. The **scsi_hba_init( )** routine must be called in the HBA's **_init**(9E) routine before **mod_install**(9F) is called. If **mod_install**(9F) fails, the HBA's **_init**(9E) should call **scsi_hba_fini**(9F) before returning failure.

scsi_hba_fini( )

scsi_hba_fini( ) is the system provided completion routine for SCSI HBA drivers. **scsi_hba_fini( )** removes all of the system references for the HBA that were created in **scsi_hba_init( )**. The **scsi_hba_fini( )** routine should be called in the HBA's **_fini**(9E) routine if **mod_remove**(9F) is successful.

RETURN VALUES

scsi_hba_init( ) returns **0** if successful, and a non-zero value otherwise. If **scsi_hba_init( )** fails, the HBA's **_init( )** entry point should return the value returned by **scsi_hba_init( )**.

CONTEXT

scsi_hba_init( ) and **scsi_hba_fini( )** should be called from **_init**(9E) or **_fini**(9E), respectively.

SEE ALSO

_init(9E), **_fini**(9E), **mod_install**(9F), **mod_remove**(9F), **scsi_pktfree**(9F), **scsi_pktalloc**(9F), **scsi_hba_tran**(9S)

Writing Device Drivers

NOTES

The HBA is responsible for ensuring that no DDI request routines are called on behalf of its SCSI target drivers once **scsi_hba_fini( )** is called.

NAME

scsi_hba_lookup_capstr – return index matching capability string

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_lookup_capstr(char **capstr***);**

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

capstr Pointer to a string.

DESCRIPTION

scsi_hba_lookup_capstr( ) attempts to match *capstr* against a known set of capability strings, and returns the defined index for the matched capability, if found.

The set of indices and capability strings is:

SCSI_CAP_DMA_MAX	"**dma-max**" or "**dma_max**"
SCSI_CAP_MSG_OUT	"**msg-out**" or "**msg_out**"
SCSI_CAP_DISCONNECT	"**disconnect**"
SCSI_CAP_SYNCHRONOUS	"**synchronous**"
SCSI_CAP_WIDE_XFER	"**wide-xfer**" or "**wide_xfer**"
SCSI_CAP_PARITY	"**parity**"
SCSI_CAP_INITIATOR_ID	"**initiator-id**"
SCSI_CAP_UNTAGGED_QING	"**untagged-qing**"
SCSI_CAP_TAGGED_QING	"**tagged-qing**"
SCSI_CAP_ARQ	"**auto-rqsense**"
SCSI_CAP_LINKED_CMDS	"**linked-cmds**"
SCSI_CAP_SECTOR_SIZE	"**sector-size**"
SCSI_CAP_TOTAL_SECTORS	"**total-sectors**"
SCSI_CAP_GEOMETRY	"**geometry**"

RETURN VALUES

scsi_hba_lookup_capstr( ) returns a non-negative index value corresponding to the capability string, or –**1** if the string does not match any known capability.

CONTEXT

scsi_hba_lookup_capstr( ) can be called from user or interrupt context.

SEE ALSO

tran_getcap(9E), **tran_setcap**(9E), **scsi_ifgetcap**(9F), **scsi_ifsetcap**(9F)
Writing Device Drivers

NAME

scsi_hba_pkt_alloc, scsi_hba_pkt_free – allocate and free a scsi_pkt structure

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_hba_pkt_alloc(dev_info_t **dip*, **struct scsi_address** **ap*, **int** *cmdlen*, **int** *statuslen*, **int** *tgtlen*, **int** *hbalen*, **int (****callback* **)(caddr_t** *arg***), caddr_t** *arg***);**

void scsi_hba_pkt_free(struct scsi_address **ap*, **struct scsi_pkt** **pkt***);**

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

dip — Pointer to a **dev_info_t** structure, defining the HBA driver instance.

ap — Pointer to a **scsi_address**(9S) structure, defining the target instance.

cmdlen — Length in bytes to be allocated for the SCSI command descriptor block (CDB).

statuslen — Length in bytes to be allocated for the SCSI status completion block (SCB).

tgtlen — Length in bytes to be allocated for a private data area for the target driver's exclusive use.

hbalen — Length in bytes to be allocated for a private data area for the HBA driver's exclusive use.

callback — indicates what **scsi_hba_pkt_alloc( )** should do when resources are not available:

- **NULL_FUNC** — Do not wait for resources. Return a **NULL** pointer.
- **SLEEP_FUNC** — Wait indefinitely for resources.

arg — Must be **NULL**.

pkt — A pointer to a **scsi_pkt**(9S) structure.

DESCRIPTION

scsi_hba_pkt_alloc( )

scsi_hba_pkt_alloc( ) allocates space for a **scsi_pkt** structure. HBA drivers should use this interface when allocating a **scsi_pkt** from their **tran_init_pkt**(9E) entry point.

If *callback* is **NULL_FUNC**, **scsi_hba_pkt_alloc( )** may not sleep when allocating resources, and callers should be prepared to deal with allocation failures.

scsi_hba_pkt_alloc( ) copies the **scsi_address**(9S) structure pointed to by *ap* to the **pkt_address** field in the **scsi_pkt**(9S).

scsi_hba_pkt_alloc( ) also allocates memory for these **scsi_pkt**(9S) data areas, and sets these fields to point to the allocated memory:

pkt_ha_private	HBA private data area
pkt_private	target driver private data area
pkt_scbp	SCSI status completion block

pkt_cdbp	SCSI command descriptor block

scsi_hba_pkt_free( )

scsi_hba_pkt_free( ) frees the space allocated for the **scsi_pkt**(9S) structure.

RETURN VALUES

scsi_hba_pkt_alloc( ) returns a pointer to the **scsi_pkt** structure, or NULL if no space is available.

CONTEXT

scsi_hba_pkt_alloc( ) can be called from user or interrupt context. Drivers must not allow **scsi_hba_pkt_alloc( )** to sleep if called from an interrupt routine.

scsi_hba_pkt_free( ) can be called from user or interrupt context.

SEE ALSO

tran_init_pkt(9E), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_hba_probe – default SCSI HBA probe function

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_hba_probe(struct scsi_device **sd,* **int (****waitfunc***)(void));**

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

sd	Pointer to a **scsi_device**(9S) structure describing the target.
waitfunc	**NULL_FUNC** or **SLEEP_FUNC**.

DESCRIPTION

scsi_hba_probe( ) is a function providing the semantics of **scsi_probe**(9F). An HBA driver may call **scsi_hba_probe( )** from its **tran_tgt_probe**(9E) entry point, to probe for the existence of a target on the SCSI bus, or the HBA may set **tran_tgt_probe**(9E) to point to **scsi_hba_probe**(9F) directly.

RETURN VALUES

See **scsi_probe**(9F) for the return values from **scsi_hba_probe( )**.

CONTEXT

scsi_hba_probe( ) should be only be called from the HBA's **tran_tgt_probe**(9E) entry point.

SEE ALSO

tran_tgt_probe(9E), **scsi_probe**(9F)

Writing Device Drivers

NAME

scsi_hba_tran_alloc, scsi_hba_tran_free – allocate and free transport structures

SYNOPSIS

#include <sys/scsi/scsi.h>

scsi_hba_tran_t ∗**scsi_hba_tran_alloc(dev_info_t** ∗*dip*, **int** *flags***);**

void scsi_hba_tran_free(scsi_hba_tran_t ∗*hba_tran***);**

INTERFACE LEVEL

Solaris architecture specific (Solaris DDI).

ARGUMENTS

dip	Pointer to a **dev_info** structure, defining the HBA driver instance.
flag	flag modifiers. The only possible flag value is **SCSI_HBA_CANSLEEP** (memory allocation may sleep).
hba_tran	Pointer to a **scsi_hba_tran**(9S) structure.

DESCRIPTION

scsi_hba_tran_alloc( )

scsi_hba_tran_alloc( ) allocates a **scsi_hba_tran**(9S) structure for a HBA driver. The HBA must use this structure to register its transport vectors with the system by using **scsi_hba_attach**(9F).

If the flag **SCSI_HBA_CANSLEEP** is set in *flags*, **scsi_hba_tran_alloc( )** may sleep when allocating resources; otherwise it may not sleep, and callers should be prepared to deal with allocation failures.

scsi_hba_tran_free( )

scsi_hba_tran_free( ) is used to free the **scsi_hba_tran**(9S) structure allocated by **scsi_hba_tran_alloc( )**.

RETURN VALUES

scsi_hba_tran_alloc( ) returns a pointer to the allocated transport structure, or NULL if no space is available.

CONTEXT

scsi_hba_tran_alloc( ) can be called from user or interrupt context. Drivers must not allow **scsi_hba_tran_alloc( )** to sleep if called from an interrupt routine.

scsi_hba_tran_free( ) can be called from user or interrupt context.

SEE ALSO

scsi_hba_attach(9F), **scsi_hba_tran**(9S)

Writing Device Drivers

NAME scsi_ifgetcap, scsi_ifsetcap − get / set SCSI transport capability

SYNOPSIS **#include <sys/scsi/scsi.h>**

int scsi_ifgetcap(struct scsi_address *∗ap,* **char** *∗cap,* **int** *whom***);**

int scsi_ifsetcap(struct scsi_address *∗ap,* **char** *∗cap,* **int** *value,* **int** *whom***);**

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

ap Pointer to the **scsi_address** structure.

cap Pointer to the string capability identifier.

value Defines the new state of the capability.

whom Determines if all targets or only the specified target is affected.

DESCRIPTION The target drivers use **scsi_ifsetcap( )** to set the capabilities of the host adapter driver. A *cap* is a name-value pair whose name is a null terminated character string and whose value is an integer. The current value of a capability can be retrieved using **scsi_ifgetcap( ).** If *whom* is **0** all targets are affected, else the target specified by the **scsi_address** structure pointed to by *ap* is affected.

A device may support only a subset of the capabilities listed below. It is the responsibility of the driver to make sure that these functions are called with a *cap* supported by the device.

The following capabilities have been defined:

“dma-max” Maximum dma transfer size supported by host adapter.

“msg-out” Message out capability supported by host adapter: 0 disables, 1 enables.

“disconnect” Disconnect capability supported by host adapter: 0 disables, 1 enables.

“synchronous” Synchronous data transfer capability supported by host adapter: 0 disables, 1 enables.

“wide-xfer” Wide transfer capability supported by host adapter: 0 disables, 1 enables.

“parity” Parity checking by host adapter: 0 disables, 1 enables.

“initiator-id” The host’s bus address is returned.

“untagged-qing” The host adapter’s capability to support internal queueing of commands without tagged queueing: 0 disables, 1 enables.

“tagged-qing” The host adapter’s capability to support tagged queuing: 0 disables, 1 enables.

“auto-rqsense” The host adapter’s capability to support auto request sense on check conditions: 0 disables, 1 enables.

“sector-size” The target driver sets this capability to inform the HBA of the granularity, in bytes, of DMA breakup; the HBA's DMA limit structure will be set to reflect this limit (See **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S)). It should be set to the physical disk sector size. This capability defaults to 512.

"total-sectors" The target driver sets this capability to inform the HBA of the total number of sectors on the device, as returned from the SCSI **get capacity** command. This capability must be set before the target driver ''gets'' the **geometry** capability.

"geometry" This capability returns the HBA geometry of a target disk. The target driver must set the **total-sectors** capability before ''getting'' the **geometry** capability. The geometry is returned as a 32-bit value: the upper 16 bits represent the number of heads per cylinder; the lower 16 bits represent the number of sectors per track. The **geometry** capability cannot be ''set.''

RETURN VALUES

scsi_ifsetcap( ) returns **1** if the capability was successfully set to the new value, **0** if the capability is not variable, and –**1** if the capability was not defined.

scsi_ifgetcap( ) returns the current value of a capability, or –**1** if the capability was not defined.

CONTEXT

These functions can be called from user or interrupt context.

EXAMPLE

```
un->un_arq_enabled =
        ((scsi_ifsetcap(&devp->sd_address, "auto-rqsense", 1, 1) == 1)? 1: 0);

if (scsi_ifsetcap(&devp->sd_address, "tagged-qing", 1, 1) == 1) {
        un->un_dp->options |= SD_QUEUEING;
        un->un_throttle = MAX_THROTTLE;
} else if (scsi_ifgetcap(&devp->sd_address, "untagged-qing", 0) == 1) {
        un->un_dp->options |= SD_QUEUEING;
        un->un_throttle = 3;
} else {
        un->un_dp->options &= ~SD_QUEUEING;
        un->un_throttle = 1;
}
```

SEE ALSO

ddi_dma_lim_sparc(9S), **ddi_dma_lim_x86**(9S), **scsi_arq_status**(9S)

Writing Device Drivers

NAME

scsi_init_pkt – prepare a complete SCSI packet

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt ∗**scsi_init_pkt(struct scsi_address** ∗*ap*, **struct scsi_pkt** ∗*pktp*,
  struct buf ∗*bp*, **int** *cmdlen*, **int** *statuslen*, **int** *privatelen*, **int** *flags*,
  int (∗*callback* **)(caddr_t), caddr_t** *arg***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ap Pointer to a **scsi_address**(9S) structure.

pktp A pointer to a **scsi_pkt**(9S) structure.

bp Pointer to a **buf**(9S) structure.

cmdlen The required length for the SCSI command descriptor block (CDB) in bytes.

statuslen The required length for the SCSI status completion block (SCB) in bytes.

privatelen The required length for the *pkt_private* area.

flags The flag for creating the packet.

callback A pointer to a callback function, **NULL_FUNC**, or **SLEEP_FUNC**.

arg The *callback* function argument.

DESCRIPTION

Target drivers use **scsi_init_pkt()** to request the transport layer to allocate and initialize a packet for a SCSI command which possibly includes a data transfer. If *pktp* is **NULL,** a new **scsi_pkt**(9S) is allocated using the HBA driver's packet allocator. The *bp* is a pointer to a **buf**(9S) structure. If *bp* is non-**NULL** and contains a valid byte count, the **buf**(9S) structure is also set up for DMA transfer using the HBA driver DMA resources allocator. When *bp* is allocated by **scsi_alloc_consistent_buf**(9F), the **PKT_CONSISTENT** bit must be set in the *flags* argument to ensure proper operation. If *privatelen* is non-zero then additional space is allocated for the *pkt_private* area of the **scsi_pkt**(9S). On return *pkt_private* points to this additional space. Otherwise *pkt_private* is a pointer that is typically used to store the *bp* during execution of the command. In this case *pkt_private* is **NULL** on return.

The *flags* argument is a set of bit flags. Possible bits include:

PKT_CONSISTENT This must be set if the DMA buffer was allocated using **scsi_alloc_consistent_buf**(9F). In this case, the HBA driver will guarantee that the data transfer is properly synchronized before performing the target driver's command completion callback.

PKT_DMA_PARTIAL This may be set if the driver can accept a partial DMA mapping. If set, **scsi_init_pkt()** will allocate DMA resources with the **DDI_DMA_PARTIAL** bit set in the **dmar_flag** element of the **ddi_dma_req**(9S) structure. The **pkt_resid** field of the **scsi_pkt**(9S) structure may be returned with a non-zero value, which indicates the number of bytes for which **scsi_init_pkt()** was unable to allocate DMA resources.

The last argument *arg* is supplied to the *callback* function when it is invoked.

callback indicates what the allocator routines should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a **NULL** pointer.

SLEEP_FUNC Wait indefinitely for resources.

Other Values *callback* points to a function which is called when resources may have become available. *callback* **must** return either **0** (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or **1** indicating either success in allocating resources or indicating that it no longer cares for a retry.

When allocating DMA resources, **scsi_init_pkt()** returns the **scsi_pkt** field **pkt_resid** as the number of residual bytes for which the system was unable to allocate DMA resources. A **pkt_resid** of **0** means that all necessary DMA resources were allocated.

RETURN VALUES

scsi_init_pkt() returns **NULL** if the packet or dma resources could not be allocated. Otherwise, it returns a pointer to an initialized **scsi_pkt**(9S). If *pktp* was not **NULL** the return value will be *pktp* on successful initialization of the packet.

CONTEXT

If *callback* is **SLEEP_FUNC,** then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block.

EXAMPLES

To allocate a packet without DMA resources attached, use:

```
pkt = scsi_init_pkt(&devp->sd_address, NULL, NULL, CDB_GROUP1,
        STATUS_LEN, sizeof (struct my_pkt_private *), 0,
        sd_runout, sd_unit);
```

To allocate a packet with DMA resources attached use:

```
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP1,
        STATUS_LEN, 0, 0, NULL_FUNC, NULL);
```

To attach DMA resources to a preallocated packet, use:

```
pkt = scsi_init_pkt(&devp->sd_address, old_pkt, bp, 0,
        0, 0, 0, sd_runout, (caddr_t) sd_unit);
```

Since the packet is already allocated the *cmdlen*, *statuslen* and *privatelen* are **0**.

To allocate a packet with consistent DMA resources attached, use:

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
        SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
        STATUS_LEN, sizeof (struct my_pkt_private *), PKT_CONSISTENT,
        SLEEP_FUNC, NULL);
```

To allocate a packet with partial DMA resources attached, use:

my_pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0, STATUS_LEN, sizeof (struct buf *), PKT_DMA_PARTIAL, SLEEP_FUNC, NULL);

NOTES

If a DMA allocation request fails with **DDI_DMA_NOMAPPING**, the **B_ERROR** flag will be set in *bp*, and the **b_error** field will be set to **EFAULT**.

If a DMA allocation request fails with **DDI_DMA_TOOBIG**, the **B_ERROR** flag will be set in *bp*, and the **b_error** field will be set to **EINVAL**.

SEE ALSO

scsi_alloc_consistent_buf(9F), **scsi_destroy_pkt**(9F), **scsi_dmaget**(9F), **scsi_pktalloc**(9F), **buf**(9S), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_log – display a SCSI-device-related message

SYNOPSIS

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

void scsi_log(dev_info_t **dip*, **char** **drv_name*, **u_int** *level*, **const char** **fmt*, **. . .);**

ARGUMENTS

dip	Pointer to the **dev_info** structure.
drv_name	String naming the device.
level	Error level.
fmt	Display format.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_log( ) is a utility function that displays a message via the **cmn_err**(9F) routine. The error levels that can be passed in to this function are **CE_PANIC, CE_WARN, CE_NOTE, CE_CONT,** and **SCSI_DEBUG.** The last level is used to assist in displaying debug messages to the console only. *drv_name* is the short name by which this device is known; example disk driver names are **sd** and **cmdk**. If the dev_info_t pointer is NULL, then the *drv_name* will be used with no unit or long name.

EXAMPLE

scsi_log(dev, "Disk Unit ", CE_PANIC, "Bad Value %d\n", foo);

Generates:
PANIC: /eisa/aha@330,0/cmdk@0,0 (Disk Unit 0): Bad Value 5
Followed by a PANIC.

scsi_log(dev, "sd", CE_WARN, "Label Bad\n");

Generates:
WARNING: /sbus@1,f8000000/esp@0,8000000/sd@1,0 (sd1): Label Bad

scsi_log((dev_info_t *) NULL, "Disk Unit ", CE_NOTE, "Disk Ejected\n");

Generates:
Disk Unit : Disk Ejected

scsi_log(cmdk_unit, "Disk Unit ", CE_CONT, "Disk Inserted\n");

Generates:
Disk Inserted

scsi_log(sd_unit, "sd", SCSI_DEBUG, "We really got here\n");

Generates (only to the console):
DEBUG: sd1: We really got here

CONTEXT

scsi_log( ) may be called from user or interrupt context.

SEE ALSO

cmn_err(9F), **scsi_errmsg**(9F)

Writing Device Drivers

NAME

scsi_pktalloc, scsi_resalloc, scsi_pktfree, scsi_resfree – SCSI packet utility routines

SYNOPSIS

#include <sys/scsi/scsi.h>

struct scsi_pkt *scsi_pktalloc(struct scsi_address **ap*, **int** *cmdlen*, **int** *statuslen*,
int (**callback***)(void));**

struct scsi_pkt *scsi_resalloc(struct scsi_address **ap*, **int** *cmdlen*, **int** *statuslen*,
opaque_t *dmatoken*, **int (****callback***)(void));**

void scsi_pktfree(struct scsi_pkt **pkt***);**

void scsi_resfree(struct scsi_pkt **pkt***);**

ARGUMENTS

ap Pointer to a **scsi_address** structure.

cmdlen The required length for the SCSI command descriptor block (CDB) in bytes.

statuslen The required length for the SCSI status completion block (SCB) in bytes.

dmatoken Pointer to an implementation-dependent object.

callback A pointer to a callback function, or **NULL_FUNC** or **SLEEP_FUNC.**

pkt Pointer to a **scsi_pkt**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_pktalloc( ) requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, **scsi_resalloc( )** should be used.

ap is a pointer to a **scsi_address** structure. Allocator routines use it to determine the associated host adapter.

cmdlen is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the **pkt_cdbp** field of the allocated **scsi_pkt** structure.

statuslen is the required length for the SCSI status completion block. The address of the allocated block is placed into the **pkt_scbp** field of the **scsi_pkt** structure.

dmatoken is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The *dmatoken* must be a pointer to a **buf**(9S) structure. If *dmatoken* is **NULL**, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data transfer phase (either no data transfer phase expected, or the wrong direction encountered), the command is terminated with the **pkt_reason** set to **CMD_DMA_DERR.** *dmatoken* provides the information to determine if the transfer count is correct.

callback indicates what the allocator routines should do when resources are not available:

NULL_FUNC Do not wait for resources. Return a **NULL** pointer.

SLEEP_FUNC Wait indefinitely for resources.

Other Values *callback* points to a function which is called when resources may have become available. *callback* **must** return either **0** (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or **1** indicating either success in allocating resources or indicating that it no longer cares for a retry.

scsi_pktfree( ) frees the packet.

scsi_resfree( ) free all resources held by the packet and the packet itself.

RETURN VALUES

Both allocation routines return a pointer to a **scsi_pkt** structure on success, or **NULL** on failure.

CONTEXT

If *callback* is **SLEEP_FUNC,** then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block. Both deallocation routines can be called from user or interrupt context.

SEE ALSO

scsi_dmafree(9F), **scsi_dmaget**(9F), **buf**(9S), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_poll – run a polled SCSI command on behalf of a target driver

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_poll(struct scsi_pkt **pkt***);**

ARGUMENTS

pkt Pointer to the **scsi_pkt**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_poll( ) requests the host adapter driver to run a polled command. Unlike **scsi_transport**(9F) which runs commands asynchronously, **scsi_poll**( ) runs commands to completion before returning. If the **pkt_time** member of *pkt* is zero it is defaulted to SCSI_POLL_TIMEOUT to prevent an indefinite hang of the system.

RETURN VALUES

scsi_poll( ) returns:

0	command completed successfully.
-1	command failed.

CONTEXT

scsi_poll ( ) can be called from user or interrupt level.

SEE ALSO

makecom(9F), **scsi_transport**(9F), **scsi_pkt**(9S)

Writing Device Drivers

WARNING

scsi_poll( ) might loop indefinitely waiting for a SCSI command to complete; hence it is not normally recommended to call it from interrupt context.

NAME

scsi_probe – utility for probing a scsi device

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_probe(struct scsi_device **devp*, **int (****waitfunc***)());**

ARGUMENTS

devp	Pointer to a **scsi_device**(9S) structure
waitfunc	**NULL_FUNC** or **SLEEP_FUNC**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_probe( ) determines whether a *target/lun* is present and sets up the **scsi_device** structure with inquiry data.

scsi_probe( ) uses the SCSI Inquiry command to test if the device exists. It may retry the Inquiry command as appropriate. If **scsi_probe( )** is successful, it will allocate space for the **scsi_inquiry** structure and assign the address to the **sd_inq** member of the **scsi_device**(9S) structure. **scsi_probe( )** will then fill in this *scsi_inquiry* structure and return **SCSIPROBE_EXISTS**.

scsi_unprobe(9F) is used to undo the effect of **scsi_probe( )**.

If the target is a non-CCS device, **SCSIPROBE_NONCCS** will be returned.

waitfunc indicates what the allocator routines should do when resources are not available; the valid values are:

NULL_FUNC	Do not wait for resources. Return **SCSIPROBE_NOMEM** or **SCSIPROBE_FAILURE**
SLEEP_FUNC	Wait indefinitely for resources.

RETURN VALUES

scsi_probe( ) returns:

SCSIPROBE_BUSY	Device exists but is currently busy.
SCSIPROBE_EXISTS	Device exists and inquiry data is valid.
SCSIPROBE_FAILURE	Polled command failure.
SCSIPROBE_NOMEM	No space available for structures.
SCSIPROBE_NONCCS	Device exists but inquiry data is not valid.
SCSIPROBE_NORESP	Device does not respond to an INQUIRY.

CONTEXT

scsi_probe( ) is normally called from the target driver's **probe**(9E) or **attach**(9E) routine. If *waitfunc* is **SLEEP_FUNC,** then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level.

EXAMPLE

```
switch (scsi_probe(devp, NULL_FUNC)) {
default:
case SCSIPROBE_NORESP:
case SCSIPROBE_NONCCS:
case SCSIPROBE_NOMEM:
case SCSIPROBE_FAILURE:
case SCSIPROBE_BUSY:
        break;

case SCSIPROBE_EXISTS:
        switch (devp->sd_inq->inq_dtype) {
        case DTYPE_DIRECT:
                rval = DDI_PROBE_SUCCESS;
                break;
        case DTYPE_RODIRECT:
                rval = DDI_PROBE_SUCCESS;
                break;
        case DTYPE_NOTPRESENT:
        default:
                break;
        }
}
scsi_unprobe(devp);
```

SEE ALSO

attach(9E), **probe**(9E), **scsi_slave**(9F), **scsi_unprobe**(9F), **scsi_unslave**(9F), **scsi_device**(9S)

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

Writing Device Drivers

NOTES

A *waitfunc* function other than **NULL_FUNC** or **SLEEP_FUNC** is not supported and may have unexpected results.

NAME

scsi_reset – reset a SCSI bus or target

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_reset(struct scsi_address **ap*, **int** *level***);**

ARGUMENTS

ap	Pointer to the **scsi_address** structure.
level	The level of reset required.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_reset( ) asks the host adapter driver to reset the SCSI bus or a SCSI target as specified by *level*. If *level* equals **RESET_ALL**, the SCSI bus is reset. If it equals **RESET_TARGET**, *ap* is used to determine the target to be reset.

RETURN VALUES

scsi_reset( ) returns:

1	on success.
0	on failure.

CONTEXT

scsi_reset ( ) can be called from user or interrupt context.

SEE ALSO

scsi_abort(9F)

Writing Device Drivers

NAME

scsi_slave – utility for SCSI target drivers to establish the presence of a target

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_slave(struct scsi_device **devp*, **int (****callback*)**(void));**

ARGUMENTS

devp Pointer to a **scsi_device**(9S) structure.

callback Pointer to a callback function, **NULL_FUNC** or **SLEEP_FUNC**.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_slave( ) checks for the presence of a SCSI device. Target drivers may use this function in their **probe**(9E) routines. **scsi_slave( )** determines if the device is present by using a Test Unit Ready command followed by an Inquiry command. If **scsi_slave( )** is successful, it will fill in the **scsi_inquiry** structure, which is the **sd_inq** member of the **scsi_device**(9S) structure, and return **SCSI_PROBE_EXISTS**. This information can be used to determine if the target driver has probed the correct SCSI device type. *callback* indicates what the allocator routines should do when DMA resources are not available:

NULL_FUNC Do not wait for resources. Return a **NULL** pointer.

SLEEP_FUNC Wait indefinitely for resources.

Other Values *callback* points to a function which is called when resources may have become available. *callback* **must** return either **0** (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or **1** indicating either success in allocating resources or indicating that it no longer cares for a retry.

RETURN VALUES

scsi_slave( ) returns:

SCSIPROBE_NOMEM	No space available for structures.
SCSIPROBE_EXISTS	Device exists and inquiry data is valid.
SCSIPROBE_NONCCS	Device exists but inquiry data is not valid.
SCSIPROBE_FAILURE	Polled command failure.
SCSIPROBE_NORESP	No response to **TEST UNIT READY.**

CONTEXT

scsi_slave( ) is normally called from the target driver's **probe**(9E) or **attach**(9E) routine. If *callback* is **SLEEP_FUNC,** then this routine may only be called from user-level code. Otherwise, it may be called from either user or interrupt level. The *callback* function may not block or call routines that block.

SEE ALSO

attach(9E), **probe**(9E), **ddi_iopb_alloc**(9F), **makecom**(9F), **scsi_dmaget**(9F), **scsi_ifgetcap**(9F), **scsi_pktalloc**(9F), **scsi_poll**(9F), **scsi_probe**(9F), **scsi_device**(9S)
ANSI Small Computer System Interface-2 (SCSI-2)
Writing Device Drivers

NAME

scsi_sync_pkt – synchronize CPU and I/O views of memory

SYNOPSIS

#include <sys/scsi/scsi.h>

void scsi_sync_pkt(struct scsi_pkt **pktp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

pktp pointer to a **scsi_pkt**(9S) structure.

DESCRIPTION

scsi_sync_pkt( ) is used to selectively synchronize a CPU's or device's view of the data associated with the SCSI packet that has been mapped for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When a SCSI packet is mapped for I/O using **scsi_init_pkt**(9F) and destroyed using **scsi_destroy_pkt**(9F), then an implicit **scsi_sync_pkt( )** will be performed. However, if the memory object has been modified by either the device or a CPU after the mapping by **scsi_init_pkt**(9F), then a call to **scsi_sync_pkt( )** is required.

EXAMPLES

If the same scsi_pkt is reused for a data transfer from memory to a device, then **scsi_sync_pkt( )** must be called before calling **scsi_transport**(9F). If the same packet is reused for a data transfer from a device to memory **scsi_sync_pkt( )** must be called after the completion of the packet but before accessing the data in memory.

CONTEXT

scsi_sync_pkt( ) may be called from user or interrupt context.

SEE ALSO

tran_sync_pkt(9E), **ddi_dma_sync**(9F), **scsi_init_pkt**(9F), **scsi_destroy_pkt**(9F), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_transport – request by a SCSI target driver to start a command

SYNOPSIS

#include <sys/scsi/scsi.h>

int scsi_transport(struct scsi_pkt **pkt***);**

ARGUMENTS

pkt Pointer to a **scsi_pkt**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

Target drivers use **scsi_transport( )** to request the host adapter driver to transport a command to the SCSI target device specified by *pkt*. The target driver must obtain resources for the packet using **scsi_init_pkt**(9F) prior to calling this function. The packet may be initialized using one of the **makecom**(9F) functions. **scsi_transport ( )** does not wait for the SCSI command to complete. See **scsi_poll**(9F) for a description of polled SCSI commands. Upon completion of the SCSI command the host adapter calls the completion routine provided by the target driver in the *pkt_comp* member of the **scsi_pkt** pointed to by *pkt*.

RETURN VALUES

scsi_transport( ) returns:

TRAN_ACCEPT	The packet was accepted by the transport layer.
TRAN_BUSY	The packet could not be accepted because there was already a packet in progress for this target/lun, the host adapter queue was full, or the target device queue was full.
TRAN_BADPKT	The DMA count in the packet exceeded the DMA engine's maximum DMA size.
TRAN_FATAL_ERROR	A fatal error has occurred in the transport layer.

CONTEXT

scsi_transport( ) can be called from user or interrupt context.

EXAMPLE

```
if ((status = scsi_transport(rqpkt)) != TRAN_ACCEPT) {
        scsi_log(devp, sd_label, CE_WARN,
            "transport of request sense pkt fails (0x%x)\n", status);
}
```

SEE ALSO

makecom(9F), **scsi_init_pkt**(9F), **scsi_pktalloc**(9F), **scsi_poll**(9F), **scsi_pkt**(9S)

Writing Device Drivers

NAME

scsi_unprobe, scsi_unslave – free resources allocated during initial probing

SYNOPSIS

#include <sys/scsi/scsi.h>

void scsi_unslave(struct scsi_device **devp***);**

void scsi_unprobe(struct scsi_device **devp***);**

ARGUMENTS

devp Pointer to a **scsi_device**(9S) structure.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

scsi_unprobe() and **scsi_unslave()** are used to free any resources that were allocated on the driver's behalf during **scsi_slave**(9F) and **scsi_probe**(9F) activity.

CONTEXT

scsi_unprobe () and **scsi_unslave ()** may be called from either the user or the interrupt levels.

SEE ALSO

scsi_probe(9F), **scsi_slave**(9F), **scsi_device**(9S)

Writing Device Drivers

NAME

semaphore, sema_init, sema_destroy, sema_p, sema_p_sig, sema_v, sema_tryp – semaphore functions

SYNOPSIS

#include <sys/ksynch.h>

void sema_init(ksema_t **sp***, u_int** *val***, char** **name***, ksema_type_t** *type***, void** **arg***);**

void sema_destroy(ksema_t **sp***);**

void sema_p(ksema_t **sp***);**

void sema_v(ksema_t **sp***);**

int sema_p_sig(ksema_t **sp***);**

int sema_tryp(ksema_t **sp***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

sp A pointer to a semaphore, type **ksema_t**.

val Initial value for semaphore.

name A string describing the semaphore for statistics and debugging.

type Variant type of the semaphore. Currently only **SEMA_DRIVER** is supported.

arg Type-specific argument, should be **NULL**.

DESCRIPTION

These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomicly decremented by **sema_p( )** and atomicly incremented by **sema_v( )**. The value must always be greater than or equal to zero. If **sema_p( )** is called and the value is zero, the calling thread is blocked until another thread performs a **sema_v( )** operation on the semaphore.

Semaphores are initialized by calling **sema_init( )**. The argument, *val*, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamicly allocated, if necessary, by **sema_init( )**. For this reason, **sema_destroy( )** should be called before deallocating the storage containing the semaphore.

sema_p_sig( ) decrements the semaphore, as does **sema_p( )**, however, if the semaphore value is zero, **sema_p_sig( )** will return without decrementing the value if a signal (e.g. from **kill**(2)) is pending for the thread.

sema_tryp( ) will decrement the semaphore value only if it is greater than zero, and will not block.

RETURN VALUES

0 **sema_tryp( )** could not decrement the semaphore value because it was zero.

1 **sema_p_sig( )** was not able to decrement the semaphore value and detected a pending signal.

CONTEXT

These function can be called from user or interrupt context, except for **sema_init( )** and **sema_destroy( )**, which can be called from user context only.

SEE ALSO

kill(2), **condvar**(9F), **mutex**(9F)

Writing Device Drivers

NAME

sprintf – format characters in memory

SYNOPSIS

#include <sys/ddi.h>

char *sprintf(char **buf*, **const char** **fmt*, *. . .);*

ARGUMENTS

buf Pointer to a character string.

fmt Pointer to a character string.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

sprintf( ) builds a string in *buf* under the control of the format *fmt*. The format is a character string with either plain characters, which are simply copied into *buf*, or conversion specifications, each of which converts zero or more arguments, again copied into *buf*. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user's responsibility to ensure that enough storage is available for *buf*.

Each conversion specification is introduced by the **%** character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional **l** (**ll**) specifying that a following **d**, **D**, **o**, **O**, **x**, **X**, or **u** conversion character applies to a **long** (**long long**) integer argument. An **l** (**ll**) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

d,**D**,**o**,**O**,**x**,**X**,**u**
The integer argument is converted to signed decimal (**d**, **D**), unsigned octal (**o**, **O**), unsigned hexadecimal (**x**, **X**) or unsigned decimal (**u**), respectively, and copied. The letters **abcdef** are used for **x** and **X** conversion.

c The character value of argument is copied.

b This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form **<***base***>[<***arg***> . . .]**. The base supplies the conversion base for the first argument as a binary value; \10 gives octal, \20 gives hexadecimal. Each subsequent <arg> is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.

A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by **<** and **>**. Thus, the following function call would generate **reg=3<BitTwo,BitOne>\n** in *buf*.

sprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")

s The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL, the string **<null string>** is used in its place.

% Copy a **%**; no argument is converted.

RETURN VALUES

sprintf( ) returns its first argument, *buf*.

CONTEXT

sprintf( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

stoi, numtos – convert between an integer and a decimal string

SYNOPSIS

#include <sys/ddi.h>

int stoi(char ***str***);**
void numtos(unsigned long *num,* **char** **s***);**

ARGUMENTS

str Pointer to a character string to be converted.

num Decimal number to be converted to a character string.

s Character buffer to hold converted decimal number.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

stoi( )

stoi( ) returns the integer value of a string of decimal numeric characters beginning at ***str*. No overflow checking is done. **str* is updated to point at the last character examined.

numtos ( )

numtos( ) converts a **long** into a null-terminated character string. No bounds checking is done. The caller must ensure there is enough space to hold the result.

RETURN VALUES

stoi( ) returns the integer value of the string *str*.

CONTEXT

stoi( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NOTES

stoi( ) handles only positive integers; it does not handle leading minus signs.

NAME strchr – find a character in a string

SYNOPSIS

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

char ∗strchr(const char ∗*str*, **int** *chr***);**

INTERFACE LEVEL Solaris DDI specific (Solaris DDI).

ARGUMENTS

str Pointer to a string to be searched.

chr The character to search for.

DESCRIPTION **strchr**( ) returns a pointer to the first occurrence of *chr* in the string pointed to by *str*.

RETURN VALUES **strchr**( ) returns a pointer to a character, or **NULL**, if the search fails.

CONTEXT This function can be called from user or interrupt context.

SEE ALSO **strcmp**(9F)

Writing Device Drivers

NAME

strcmp, strncmp – compare two null terminated strings.

SYNOPSIS

#include <sys/ddi.h>

int strcmp(const char **s1,* **const char** **s2***);**
int strncmp(const char **s1,* **const char** **s2,* **size_t** *n***);**

ARGUMENTS

s1, s2 Pointers to character strings.

n Count of characters to be compared.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

strcmp( )

strcmp( ) returns **0** if the strings are the same, or the integer value of the expression *(*s1 - *s2)* for the last characters compared if they differ.

strncmp( )

strncmp( ) returns **0** if the first *n* characters of *s1* and *s2* are the same, or *(*s1 - *s2)* for the last characters compared if they differ.

RETURN VALUES

strcmp( ) returns **0** if the strings are the same, or *(*s1 - *s2)* for the last characters compared if they differ.

strncmp( ) returns **0** if the first *n* characters of strings are the same, or *(*s1 - *s2)* for the last characters compared if they differ.

CONTEXT

These functions can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

strcpy, strncpy – copy a string from one location to another.

SYNOPSIS

#include <sys/ddi.h>

char ∗**strcpy(char** ∗*dst***, char** ∗*srs***);**

char ∗**strncpy(char** ∗*dst***, char** ∗*srs***, size_t** *n* **);**

ARGUMENTS

dst, srs Pointers to character strings.

n Count of characters to be copied.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

strcpy( )

strcpy( ) copies characters in the string *srs* to *dst*, terminating at the first null character in *srs*, and returns *dst* to the caller. No bounds checking is done.

strncpy( )

strncpy( ) copies *srs* to *dst*, null-padding or truncating at *n* bytes, and returns *dst*. No bounds checking is done.

RETURN VALUES

strcpy( ), and **strncpy**( ) return *dst*.

CONTEXT

strcpy( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

strlen – determine the number of non-null bytes in a string.

SYNOPSIS

#include <sys/ddi.h>

size_t strlen(const char ∗*s***);**

ARGUMENTS

s Pointer to a character string.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

strlen( ) returns the number of non-null bytes in the string argument *s*.

RETURN VALUES

strlen( ) returns the number of non-null bytes in *s*.

CONTEXT

strlen( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

NAME

strlog – submit messages to the log driver

SYNOPSIS

#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/log.h>

int strlog(short *mid***, short** *sid***, char** *level***, unsigned short** *flags***, char** **fmt***, ...** *);*

ARGUMENTS

mid Identification number of the module or driver submitting the message (in the case of a module, its **mi_idnum** value from **module_info**(9S)).

sid Identification number for a particular minor device.

level Tracing level for selective screening of low priority messages. Larger values imply less important information.

flags Valid flag values are:

SL_ERROR	Message is for error logger.
SL_TRACE	Message is for trace.
SL_NOTIFY	Mail copy of message to system administrator.
SL_CONSOLE	Log message to console.
SL_FATAL	Error is fatal.
SL_WARN	Error is a warning.
SL_NOTE	Error is a notice.

fmt **printf**(3S) style format string. **%s**, **%e**, **%g**, and **%G** formats are not allowed.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

strlog( ) submits formatted messages to the **log**(7) driver. The messages can be retrieved with the **getmsg**(2) system call. The *flags* argument specifies the type of the message and where it is to be sent. **strace**(1M) receives messages from the **log** driver and sends them to the standard output. **strerr**(1M) receives error messages from the **log** driver and appends them to a file called **/var/adm/streams/error.** *mm-dd*, where *mm-dd* identifies the date of the error message.

RETURN VALUES

strlog( ) returns **0** if the message is not seen by all the readers, **1** otherwise.

CONTEXT

strlog( ) can be called from user or interrupt context.

SEE ALSO

strace(1M), **strerr**(1M), **getmsg**(2), **log**(7), **module_info**(9S)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

strqget – get information about a queue or band of the queue

SYNOPSIS

#include <sys/stream.h>

int strqget(queue_t ∗*q*, **qfields_t** *what*, **unsigned char** *pri*, **long** ∗*valp***);**

ARGUMENTS

q Pointer to the queue.

what Field of the **queue** structure for (or the specified priority band) to return information about. Valid values are one of:

QHIWAT	High water mark.
QLOWAT	Low water mark.
QMAXPSZ	Largest packet accepted.
QMINPSZ	Smallest packet accepted.
QCOUNT	Approximate size (in bytes) of data.
QFIRST	First message.
QLAST	Last message.
QFLAG	Status.

pri Priority band of interest.

valp The address of where to store the value of the requested field.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

strqget( ) gives drivers and modules a way to get information about a queue or a particular band of a queue without directly accessing STREAMS data structures, thus insulating them from changes in the implementation of these data structures from release to release.

RETURN VALUES

On success, **0** is returned and the value of the requested field is stored in the location pointed to by *valp*. An error number is returned on failure.

CONTEXT

strqget( ) can be called from user or interrupt context.

SEE ALSO

freezestr(9F), **queue**(9S), **strqset**(9F), **unfreezestr**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES

The stream must be frozen using **freezestr**(9F) before calling **strqget**( ).

NAME

strqset – change information about a queue or band of the queue

SYNOPSIS

#include <sys/stream.h>

int strqset(queue_t **q*, **qfields_t** *what*, **unsigned char** *pri*, **long** *val***);**

ARGUMENTS

q Pointer to the queue.

what Field of the **queue** structure (or the specified priority band) to return information about. Valid values are one of:

- **QHIWAT** High water mark.
- **QLOWAT** Low water mark.
- **QMAXPSZ** Largest packet accepted.
- **QMINPSZ** Smallest packet accepted.

pri Priority band of interest.

val The value for the field to be changed.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

strqset( ) gives drivers and modules a way to change information about a queue or a particular band of a queue without directly accessing STREAMS data structures.

RETURN VALUES

On success, **0** is returned. **EINVAL** is returned if an undefined attribute is specified.

CONTEXT

strqset( ) can be called from user or interrupt context.

SEE ALSO

freezestr(9F), **queue**(9S), **strqget**(9F), **unfreezestr**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES

The stream must be frozen using **freezestr**(9F) before calling **strqset**( ).

To set the values of **QMINPSZ** and **QMAXPSZ** from within a single call to **freezestr**(9F) / **unfreezestr**(9F): when lowering the existing values, set **QMINPSZ** before setting **QMAXPSZ**; when raising the existing values, set **QMAXPSZ** before setting **QMINPSZ**.

NAME

swab – swap bytes in 16-bit halfwords

SYNOPSIS

#include <sys/sunddi.h>

void swab (void ∗*src***, void** ∗*dst***, size_t** *nbytes* **);**

ARGUMENTS

src A pointer to the buffer containing the bytes to be swapped.

dst A pointer to the destination buffer where the swapped bytes will be written. If *dst* is the same as *src* the buffer will be swapped in place.

nbytes Number of bytes to be swapped, rounded down to the nearest half-word.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

swab( ) copies the bytes in the buffer pointed to by *src* to the buffer pointer to by *dst*, swapping the order of adjacent bytes in half-word pairs as the copy proceeds. A total of *nbytes* bytes are copied, rounded down to the nearest half-word.

CONTEXT

swab( ) can be called from user or interrupt context.

NOTES

Since **swab**( ) operates byte-by-byte, it can be used on non-aligned buffers.

SEE ALSO

Writing Device Drivers

NAME

testb – check for an available buffer

SYNOPSIS

#include <sys/stream.h>

int testb(int *size*, **unsigned int** *pri*);

ARGUMENTS

size	Size of the requested buffer.
pri	Priority of the allocb request.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

testb( ) checks to see if an **allocb**(9F) call is likely to succeed if a buffer of *size* bytes at priority *pri* is requested. Even if **testb( )** returns successfully, the call to **allocb** (9F) can fail. The *pri* argument is no longer used, but is retained for compatibility.

RETURN VALUE

Returns **1** if a buffer of the requested size is available, and **0** if one is not.

CONTEXT

testb( ) can be called from user or interrupt context.

EXAMPLES

In a service routine, if **copymsg**(9F) fails (line 6), the message is put back on the queue (line 7) and a routine, **tryagain**, is scheduled to be run in one tenth of a second. Then the service routine returns.

When the **timeout**(9F) function runs, if there is no message on the front of the queue, it just returns. Otherwise, for each message block in the first message, check to see if an allocation would succeed. If the number of message blocks equals the number we can allocate, then enable the service procedure. Otherwise, reschedule **tryagain** to run again in another tenth of a second. Note that **tryagain** is merely an approximation. Its accounting may be faulty. Consider the case of a message comprised of two 1024-byte message blocks. If there is only one free 1024-byte message block and no free 2048-byte message blocks, then **testb( )** will still succeed twice. If no message blocks are freed of these sizes before the service procedure runs again, then the **copymsg**(9F) will still fail. The reason **testb( )** is used here is because it is significantly faster than calling **copymsg**. We must minimize the amount of time spent in a **timeout** routine.

```
1  xxxsrv(q)
2     queue_t *q;
3  {
4        mblk_t *mp;
5        mblk_t *nmp;
      . . .
6        if ((nmp = copymsg(mp)) == NULL) {
7                 putbq(q, mp);
8                 timeout(tryagain, (long)q, drv_usectohz(100000));
9                 return;
10       }
         . . .
```

```
}

tryagain(q)
    queue_t *q;
{
        register int can_alloc = 0;
        register int num_blks = 0;
        register mblk_t *mp;

        if (!q->q_first)
                return;
        for (mp = q->q_first; mp; mp = mp->b_cont) {
                num_blks++;
                can_alloc += testb((mp->b_datap->db_lim -
                    mp->b_datap->db_base), BPRI_MED);
        }
        if (num_blks == can_alloc)
                qenable(q);
        else
                timeout(tryagain, (long)q, drv_usectohz(100000));
}
```

SEE ALSO

allocb(9F), **bufcall**(9F), **copymsg**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NOTES

The *pri* argument is provided for compatibility only. Its value is ignored.

NAME

timeout – execute a function after a specified length of time

SYNOPSIS

#include <sys/types.h>

int timeout(void *(*func)***(caddr_t), caddr_t** *arg***, long** *ticks***);**

ARGUMENTS

func Kernel function to invoke when the time increment expires.

arg Argument to the function.

ticks Number of clock ticks to wait before the function is called.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

The **timeout( )** function schedules the specified function to be called after a specified time interval. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.

The function called by **timeout( )** must adhere to the same restrictions as a driver soft interrupt handler.

The **timeout( )** function returns an identifier that may be passed to the **untimeout**(9F) function to cancel a pending request.

RETURN VALUES

Under normal conditions, **timeout( )** returns an integer timeout identifier not equal to zero. If, however, the timeout table is full, the system will panic with the following panic message:

PANIC: Timeout table overflow

CONTEXT

timeout( ) can be called from user or interrupt context.

EXAMPLE

In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 minutes, the device driver will print out an error message to the console.

```
static void
xxtimeout_handler(caddr_t arg)
{
        struct xxstate *xsp = (struct xxstate *)arg;

        mutex_enter(&xsp->lock);
        cv_signal(&xsp->cv);
        xsp->timeout_id = 0;
        xsp->flags |= TIMED_OUT;
        mutex_exit(&xsp->lock);
}
```

```
static u_int
xxintr(caddr_t arg)
{
        struct xxstate *xsp = (struct xxstate *)arg;
    .
    .
    .
        mutex_enter(&xsp->lock);
        if (xsp->timeout_id != 0) {
                (void) untimeout(xsp->timeout_id);
                xsp->timeout_id = 0;
        }

        /* Service interrupt */

        cv_signal(&xsp->cv);
        mutex_exit(&xsp->lock);

        return(DDI_INTR_CLAIMED);
}

static void
xxcheckcond(struct xxstate *xsp)
{
    .
    .
    .
        mutex_enter(&xsp->lock);
        xsp->timeout_id = timeout(xxtimeout_handler,
          (caddr_t)xsp, (5 * drv_usectohz(1000000));
        while (/* Waiting for interrupt  or timeout*/)
                cv_wait(&xsp->cv, &xsp->lock);

        if (xsp->flags & TIMED_OUT)
                cmn_err(CE_WARN, "Device not responding");
    .
    .
    .
        mutex_exit(&xsp->lock);
    .
    .
    .
}
```

SEE ALSO

bufcall(9F), **delay**(9F), **untimeout**(9F)

Writing Device Drivers

NAME

uiomove – copy kernel data using uio structure

SYNOPSIS

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

int uiomove(caddr_t *address,* **long** *nbytes,* **enum uio_rw** *rwflag,* **uio_t** ∗*uio_p***);**

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

ARGUMENTS

address Source / destination kernel address of the copy.

nbytes Number of bytes to copy.

rwflag Flag indicating read or write operation. Possible values are **UIO_READ** and **UIO_WRITE**.

uio_p Pointer to the **uio** structure for the copy.

DESCRIPTION

The **uiomove( )** function copies *nbytes* of data to or from the space defined by the **uio** structure (described in **uio.h**) and the driver.

The **uio_segflg** member of the **uio**(9S) structure determines the the type of space to or from which the transfer being made. If it is set to **UIO_SYSSPACE** the data transfer is between addresses in the kernel. If it is set to **UIO_USERSPACE** the transfer is between a user program and kernel space.

In addition to moving the data, **uiomove( )** adds the number of bytes moved to the **iov_base** member of the **iovec**(9S) structure, decreases the **iov_len** member, increases the **uio_offset** member of the **uio**(9S) structure, and decreases the **uio_resid** member.

This function does automatic page boundary checking. *nbytes* does not have to be word-aligned.

RETURN VALUES

uiomove( ) returns 0 upon success or **EFAULT** on failure.

CONTEXT

User context only, if **uio_segflg** is set to **UIO_USERSPACE**. User or interrupt context, if **uio_segflg** is set to **UIO_SYSSPACE**.

SEE ALSO

ureadc(9F), **uwritec**(9F), **iovec**(9S), **uio**(9S)

Writing Device Drivers

WARNINGS

If **uio_segflg** is set to **UIO_SYSSPACE** and *address* is selected from user space, the system may panic.

NAME

unbufcall – cancel a pending bufcall request

SYNOPSIS

#include <sys/stream.h>

void unbufcall(int *id***);**

ARGUMENTS

id Identifier returned from **bufcall**(9F) or **esbbcall**(9F)

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

unbufcall cancels a pending **bufcall**( ) or **esbbcall**( ) request. The argument *id* is a non-zero identifier for the request to be cancelled. *id* is returned from the **bufcall**( ) or **esbbcall**( ) function used to issue the request.

unbufcall( ) will not return until the pending callback is cancelled or has run. Because of this, locks acquired by the callback routine should not be held across the call to **unbufcall**( ) or deadlock may result.

RETURN VALUES

None.

CONTEXT

unbufcall( ) can be called from user or interrupt context.

SEE ALSO

bufcall(9F), **esbbcall**(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

unlinkb – remove a message block from the head of a message

SYNOPSIS

#include <sys/stream.h>

mblk_t *unlinkb(mblk_t ***mp*****);**

ARGUMENTS

mp Pointer to the message.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

unlinkb( ) removes the first message block from the message pointed to by *mp*. A new message, minus the removed message block, is returned.

RETURN VALUES

If successful, **unlinkb( )** returns a pointer to the message with the first message block removed. If there is only one message block in the message, **NULL** is returned.

CONTEXT

unlinkb( ) can be called from user or interrupt context.

EXAMPLE

The routine expects to get passed an **M_PROTO T_DATA_IND** message. It will remove and free the **M_PROTO** header and return the remaining **M_DATA** portion of the message.

```
mblk_t *
makedata(mp)
    mblk_t *mp;
{
        mblk_t *nmp;

        nmp = unlinkb(mp);
        freeb(mp);
        return(nmp);
}
```

SEE ALSO

linkb(9F)

Writing Device Drivers
STREAMS Programmer's Guide

NAME

untimeout – cancel previous timeout function call

SYNOPSIS

#include <sys/types.h>

int untimeout(int *id***);**

ARGUMENTS

id Identification value generated by a previous **timeout**(9F) function call.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

untimeout( ) cancels a pending **timeout**(9F) request. **untimeout**( ) will not return until the pending callback is cancelled or has run. Because of this, locks acquired by the callback routine should not be held across the call to **untimeout**( ) or a deadlock may result.

RETURN VALUES

untimeout( ) returns **-1** if the *id* is not found. Otherwise, it returns an integer value greater than or equal to **0**.

CONTEXT

untimeout( ) can be called from user or interrupt context.

EXAMPLE

In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 minutes, the device driver will print out an error message to the console.

```
static void
xxtimeout_handler(caddr_t arg)
{
	struct xxstate *xsp = (struct xxstate *)arg;

	mutex_enter(&xsp->lock);
	cv_signal(&xsp->cv);
	xsp->timeout_id = 0;
	xsp->flags |= TIMED_OUT;
	mutex_exit(&xsp->lock);
}
static u_int
xxintr(caddr_t arg)
{
	struct xxstate *xsp = (struct xxstate *)arg;
	 .
	 .
	 .
	mutex_enter(&xsp->lock);
	if (xsp->timeout_id != 0) {
		(void) untimeout(xsp->timeout_id);
		xsp->timeout_id = 0;
	}
```

```
        /* Service interrupt */

        cv_signal(&xsp->cv);
        mutex_exit(&xsp->lock);

        return(DDI_INTR_CLAIMED);
}

static void
xxcheckcond(struct xxstate *xsp)
{
        .
        .
        .
        mutex_enter(&xsp->lock);
        xsp->timeout_id = timeout(xxtimeout_handler,
            (caddr_t)xsp, (5 * drv_usectohz(1000000));
        while (/* Waiting for interrupt  or timeout*/)
                cv_wait(&xsp->cv, &xsp->lock);

        if (xsp->flags & TIMED_OUT)
                cmn_err(CE_WARN, "Device not responding");
        .
        .
        .
        mutex_exit(&xsp->lock);
        .
        .
        .
}
```

SEE ALSO

open(9E), **cv_signal**(9F), **cv_wait_sig**(9F), **delay**(9F), **timeout**(9F)

Writing Device Drivers

NAME

ureadc – add character to a uio structure

SYNOPSIS

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

int ureadc(int *c***, uio_t** **uio_p***);**

ARGUMENTS

c The character added to the **uio** (9S) structure.

uio_p Pointer to the **uio**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

ureadc( ) transfers the character *c* into the address space of the **uio**(9S) structure pointed to by *uio_p*, and updates the **uio** structure as for **uiomove**(9F).

RETURN VALUES

0 is returned on success and **EFAULT** on failure.

CONTEXT

ureadc( ) can be called from user or interrupt context.

SEE ALSO

uiomove(9F), **uwritec**(9F), **iovec**(9S), **uio**(9S)

Writing Device Drivers

NAME

uwritec – remove a character from a uio structure

SYNOPSIS

#include <sys/uio.h>

int uwritec (uio_t ∗*uio_p***);**

ARGUMENTS

uio_p Pointer to the **uio**(9S) structure.

INTERFACE LEVEL

Architecture independent level 1 (DDI / DKI).

DESCRIPTION

uwritec( ) returns a character from the **uio** structure pointed to by *uio_p*, and updates the **uio** structure as for **uiomove**(9F).

RETURN VALUES

The next character for processing is returned on success, and **-1** is returned if **uio** is empty or there is an error.

CONTEXT

uwritec( ) can be called from user or interrupt context.

SEE ALSO

uiomove(9F), **ureadc**(9F), **iovec**(9S), **uio**(9S)

Writing Device Drivers

NAME

vsprintf – format characters in memory

SYNOPSIS

#include <sys/ddi.h>

char ∗**vsprintf(char** ∗*buf***, const char** ∗*fmt***, va_list** *ap***);**

ARGUMENTS

buf　Pointer to a character string.

fmt　Pointer to a character string.

ap　Pointer to a variable argument list.

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

vsprintf( ) builds a string in *buf* under the control of the format *fmt*. The format is a character string with either plain characters, which are simply copied into *buf*, or conversion specifications, each of which converts zero or more arguments, again copied into *buf*. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user's responsibility to ensure that enough storage is available for *buf*.

Each conversion specification is introduced by the **%** character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional **l** (**ll**) specifying that a following **d**, **D**, **o**, **O**, **x**, **X**, or **u** conversion character applies to a **long** (**long long**) integer argument. An **l** (**ll**) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

d,**D**,**o**,**O**,**x**,**X**,**u**
The integer argument is converted to signed decimal (**d**, **D**), unsigned octal (**o**, **O**), unsigned hexadecimal (**x**, **X**) or unsigned decimal (**u**), respectively, and copied. The letters **abcdef** are used for **x** and **X** conversion.

c　The character value of argument is copied.

b　This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form **<***base***>[<***arg***> . . .]**. The base supplies the conversion base for the first argument as a binary value; \10 gives octal, \20 gives hexadecimal. Each subsequent <arg> is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.

A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by **<** and **>**. Thus, the following function call would generate **reg=3<BitTwo,BitOne>\n** in *buf*.

vsprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")

s The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL, the string **<null string>** is used in its place.

% Copy a **%**; no argument is converted.

RETURN VALUES

vsprintf( ) returns its first argument, *buf*.

CONTEXT

vsprintf( ) can be called from user or interrupt context.

SEE ALSO

Writing Device Drivers

Index

A

add a fully initialized kstat to the system — `kstat_install`, 9F-194
`adjmsg` — trim bytes from a message, 9F-35
allocate and free a scsi_pkt structure — `scsi_hba_pkt_alloc`, 9F-274
scsi_hba_pkt_free, 9F-274
allocate and free non-sequentially accessed memory — `ddi_iopb_alloc`, 9F-123
— `ddi_iopb_free`, 9F-123
allocate and free transport structures — `scsi_hba_tran_alloc`, 9F-277
scsi_hba_tran_free, 9F-277
allocate space — `rmalloc`, 9F-246
allocate space from a resource map — `rmalloc_wait`, 9F-249
`assert` — expression verification, 9F-5
asynchronous STREAMS perimeter upgrade — `qwriter`, 9F-245

B

`bcopy` — copy data between address locations in kernel, 9F-41
`bioerror` — indicate error in buffer header, 9F-45
`bioreset` — reuse a private buffer header after I/O is complete, 9F-46
`bufcall` — call a function when a buffer becomes available, 9F-314, 9F-53
buffer header
indicate error — `bioerror`, 9F-45
reuse a private buffer header after I/O is complete — `bioreset`, 9F-46
busy-wait for specified interval — `drv_usecwait`, 9F-161
byte streams
compare two — `bcmp`, 9F-40
bytes, size
convert size in pages — `ptob`, 9F-221
convert to size in memory pages (round down) — `btop`, 9F-51
convert to size in memory pages (round up) — `btopr`, 9F-52

C

call a function when a buffer becomes available — `qbufcall`, 9F-234
call a STREAMS put procedure — `put`, 9F-224
cancel a pending qbufcall request — `qunbufcall`, 9F-241
cancel previous timeout function call — `quntimeout`, 9F-242
character strings
compare two null terminated strings — `strcmp, strncmp`, 9F-301
convert between an integer and a decimal

character strings, *continued*
— stoi, numtos, 9F-299
copy a string from one location to another — strcpy, strncpy, 9F-302
determine the number of non-null bytes in a string — strlen, 9F-303
find a character in a string — strchr, 9F-300
format in memory — sprintf, 9F-297
check for an available buffer — testb, 9F-308
condition variable routines, driver
— condvar, 9F-63
— cv_broadcast, 9F-63
— cv_init, 9F-63
— cv_signal, 9F-63
— cv_timedwait, 9F-63
— cv_timedwait_sig, 9F-63
— cv_wait, 9F-63
— cv_wait_sig, 9F-63
control driver notification of user accesses — ddi_mapdev_intercept, 9F-129
ddi_mapdev_nointercept, 9F-129
convert a DMA segment to a DMA address cookie — ddi_dma_segtocookie, 9F-104
convert clock ticks to microseconds — drv_hztousec, 9F-158
convert microseconds to clock ticks — drv_usectohz, 9F-160
create a minor node for this device — ddi_create_minor_node, 9F-84
create and initialize a new kstat — kstat_create, 9F-192
create driver-controlled mapping of device — ddi_mapdev, 9F-128

D

datamsg — test whether a message is a data message, 9F-74
DDI access credential structure
— ddi_get_cred, 9F-116
DDI announce a device
— ddi_report_dev, 9F-144
DDI device access
DDI device access, *continued*
slave access only — ddi_slaveonly, 9F-147
DDI device critical region of control
enter — ddi_enter_critical, 9F-114
exit — ddi_exit_critical, 9F-114
DDI device information structure
find parent — ddi_get_parent, 9F-120
get the root of the dev_info tree — ddi_root_node, 9F-145
remove a minor node for this devinfo — ddi_remove_minor_node, 9F-143
DDI device instance number
get — ddi_get_instance, 9F-118
DDI device mapping
ddi_mapdev — create driver-controlled mapping of device, 9F-128
ddi_mapdev_intercept — control driver notification of user accesses, 9F-129
ddi_mapdev_nointercept — control driver notification of user accesses, 9F-129
DDI device registers
map — ddi_map_regs, 9F-125
return the number of register sets — ddi_dev_nregs, 9F-89
return the size — ddi_dev_regsize, 9F-90
unmap — ddi_unmap_regs, 9F-125
DDI device virtual address
read a character — ddi_peekc, 9F-133
read a long — ddi_peekl, 9F-133
read a longlong — ddi_peekd, 9F-133
read a short — ddi_peeks, 9F-133
read a value — ddi_peek, 9F-133
write a character — ddi_pokec, 9F-135
write a long — ddi_pokel, 9F-135
write a longlong — ddi_poked, 9F-135
write a short — ddi_pokes, 9F-135
write a value — ddi_poke, 9F-135
DDI device's private data area
get the address — ddi_get_driver_private, 9F-117
set the address — ddi_set_driver_private, 9F-117
DDI devinfo node name
return — ddi_get_name, 9F-119
DDI direct memory access

DDI direct memory access, *continued*
convert DMA handle to DMA addressing cookie — ddi_dma_htoc, 9F-98
DDI direct memory access services
allocate consistent memory— ddi_iopb_alloc, 9F-131
convert a DMA cookie — ddi_dma_coff, 9F-94
easier DMA setup — ddi_dma_addr_setup, 9F-91, 9F-92
find minimum alignment and transfer size for device — ddi_iomin, 9F-122
find post DMA mapping alignment and minimum effect properties — ddi_dma_devalign, 9F-96
free consistent memory — ddi_iopb_free, 9F-131
report current DMA window offset and size — ddi_dma_curwin, 9F-95
setup DMA mapping — ddi_dma_setup, 9F-101, 9F-102, 9F-104
setup DMA resources — ddi_dma_setup, 9F-106
shift current DMA window — ddi_dma_movwin, 9F-99
tear down DMA mapping — ddi_dma_free, 9F-97
DDI interrupt handling
add an interrupt — ddi_add_intr, 9F-75
indicate interrupt handler type — ddi_intr_hilevel, 9F-121
remove an interrupt — ddi_remove_intr, 9F-75
return the number of interrupt specifications — ddi_dev_nintrs, 9F-88
DDI memory mapping
map a segment — ddi_segmap, 9F-146
DDI page size conversions
— ddi_btop, 9F-79
— ddi_btopr, 9F-79
— ddi_ptob, 9F-79
DDI property management
create properties for leaf device drivers — ddi_prop_create, 9F-137
— ddi_getlongprop, 9F-141
DDI property management, *continued*
— ddi_getlongprop_buf, 9F-141
— ddi_getprop, 9F-141
— ddi_getproplen, 9F-141
— ddi_prop_op, 9F-141
modify properties for leaf device drivers — ddi_prop_modify, 9F-137
remove all properties for leaf device drivers — ddi_prop_remove_all, 9F-137
remove properties for leaf device drivers — ddi_prop_remove, 9F-137
DDI self identifying devices
tell whether a device is self-identifying — ddi_dev_is_sid, 9F-86
DDI soft interrupt handling
add a soft interrupt — ddi_add_softintr, 9F-77
remove a soft interrupt — ddi_remove_softintr, 9F-77
trigger a soft interrupt — ddi_trigger_softintr, 9F-77
DDI soft state utility routines
allocate state structure — ddi_soft_state_zalloc, 9F-148
— ddi_soft_state, 9F-148
free soft state entry — ddi_soft_state_free, 9F-148
get pointer to soft state — ddi_get_soft_state, 9F-148
initialize state — ddi_soft_state_init, 9F-148
remove all state info — ddi_soft_state_fini, 9F-148
ddi_create_minor_node — create a minor node for this device, 9F-84
ddi_dma_burstsizes — find out the allowed burst sizes for a DMA mapping, 9F-93
ddi_dma_nextseg — get next DMA segment, 9F-101
ddi_dma_nextwin — get next DMA window, 9F-102
ddi_dma_segtocookie — convert a DMA segment to a DMA address cookie, 9F-104
ddi_dma_sync — synchronize CPU and I/O views

of memory, 9F-108
ddi_dmae — system DMA engine functions, 9F-110
ddi_dmae_1stparty — system DMA engine functions, 9F-110
ddi_dmae_alloc — system DMA engine functions, 9F-110
ddi_dmae_disable — system DMA engine functions, 9F-110
ddi_dmae_enable — system DMA engine functions, 9F-110
ddi_dmae_getcnt — system DMA engine functions, 9F-110
ddi_dmae_getlim — system DMA engine functions, 9F-110
ddi_dmae_prog — system DMA engine functions, 9F-110
ddi_dmae_release — system DMA engine functions, 9F-110
ddi_dmae_stop — system DMA engine functions, 9F-110
ddi_ffs — find first (last) bit set in a long integer, 9F-115
ddi_fls — find first (last) bit set in a long integer, 9F-115
ddi_iopb_alloc — allocate and free non-sequentially accessed memory, 9F-123
ddi_iopb_free — allocate and free non-sequentially accessed memory, 9F-123
ddi_mapdev — create driver-controlled mapping of device, 9F-128
ddi_mapdev_intercept — control driver notification of user accesses, 9F-129
ddi_mapdev_intercept — control driver notification of user accesses, 9F-129
default SCSI HBA probe function — scsi_hba_probe, 9F-276
delay — delay process execution for a specified number of clock ticks, 9F-153
Device Driver Interface, See DDI
device switch tables
return function for insignificant entries — nulldev, 9F-214
devices
get major device number — getmajor, 9F-180
get minor device number — getminor, 9F-181
make device number from major and minor numbers — makedevice, 9F-201
devices, non-pollable
error return function — nochpoll, 9F-211
disksort — single direction elevator seek sort for buffers, 9F-155
DMA mapping, the allowed burst sizes for — ddi_dma_burstsizes, 9F-93
driver buffers
copy data— copyin, 9F-80
copy data from driver — copyout, 9F-82
copy data from driver to user program — copyout, 9F-72
copy data from user program — copyin, 9F-68
return to free list — brelse, 9F-50
driver error messages
display an error message or panic the system — cmn_err, 9F-61
driver privilege — drv_priv, 9F-159
drv_getparm — retrieve kernel state information, 9F-156
drv_hztousec — convert clock ticks to microseconds, 9F-158
drv_priv — determine driver privilege, 9F-159
drv_usectohz — convert microseconds to clock ticks, 9F-160
drv_usecwait — busy-wait for specified interval, 9F-161
dupb — duplicate a message block descriptor, 9F-162
duplicate a message — dupmsg, 9F-165
duplicate a message block descriptor — dupb, 9F-162
dupmsg — duplicate a message, 9F-165

E

error return function for illegal entries — nodev, 9F-212
expression verification

expression verification, *continued*
— assert, 9F-5

F

find first (last) bit set in a long integer — ddi_ffs, 9F-115
ddi_fls, 9F-115
flushband — flush messages for specified priority band, 9F-170
free space — rmfree, 9F-251
freerbuf — free a raw buffer header, 9F-175
freeze, thaw the state of a stream — freezestr, 9F-176
unfreezestr, 9F-176
freezestr — freeze, thaw the state of a stream, 9F-176

G

get next DMA segment — ddi_dma_nextseg, 9F-101
get next DMA window — ddi_dma_nextwin, 9F-102
getmajor — get major device number, 9F-180
getminor — get minor device number, 9F-181
getrbuf — get a raw buffer header, 9F-183

I

I/O error
return — geterror, 9F-179
I/O, block
suspend processes pending completion — biowait, 9F-47
I/O, buffer
release buffer and notify processes — biodone, 9F-43
I/O, paged request
allocate virtual address space — bp_mapin, 9F-48
deallocate virtual address space — bp_mapout, 9F-49
I/O, physical
— minphys, 9F-216
— physio, 9F-216
inb — read from an I/O port, 9F-185
initialize a named kstat — kstat_named_init, 9F-195
inl — read from an I/O port, 9F-185
inw — read from an I/O port, 9F-185

K

kernel address locations
between locations — bcopy, 9F-41
kernel addresses
get page frame number — hat_getkpfnum, 9F-184
kernel memory
allocate and clear space — kmem_zalloc, 9F-190
allocate space — kmem_alloc, 9F-188
free previously allocated — kmem_free, 9F-189
kernel modules, dynamic loading
add loadable module — mod_install, 9F-204
query loadable module — mod_info, 9F-204
remove loadable module — mod_remove, 9F-204
kstat_create — create and initialize a new kstat, 9F-192
kstat_delete — remove a kstat from the system, 9F-193
kstat_install — add a fully initialized kstat to the system, 9F-194
kstat_named_init — initialize a named kstat, 9F-195
kstat_queue — update I/O kstat statistics, 9F-196
kstat_runq_back_to_waitq — update I/O kstat statistics, 9F-196
kstat_runq_enter — update I/O kstat statistics, 9F-196
kstat_runq_exit — update I/O kstat statistics, 9F-196
kstat_waitq_enter — update I/O kstat statistics, 9F-196
kstat_waitq_exit — update I/O kstat statistics, 9F-196
kstat_waitq_to_runq — update I/O kstat statis-

tics, 9F-196

M

makedevice — make device number from major and minor numbers, 9F-201
max — return the larger of two integers, 9F-202
memory
clear for a given number of bytes — bzero, 9F-56
min — return the lesser of two integers, 9F-203
minor node for device
create — ddi_create_minor_node, 9F-84
mt-streams — STREAMS multithreading, 9F-207
mutex routines
— mutex, 9F-209
— mutex_destroy, 9F-209
— mutex_enter, 9F-209
— mutex_exit, 9F-209
— mutex_init, 9F-209
— mutex_owned, 9F-209
— mutex_tryenter, 9F-209
mutual exclusion lock, See mutex

O

OTHERQ — get pointer to queue's partner queue, 9F-31
outb — write to an I/O port, 9F-215
outl — write to an I/O port, 9F-215
outw — write to an I/O port, 9F-215

P

pollwakeup — inform a process that an event has occurred, 9F-218
proc_ref — send a signal to a process, 9F-219
proc_signal — send a signal to a process, 9F-219
proc_unref — send a signal to a process, 9F-219
put — call a STREAMS put procedure, 9F-224

Q

qbufcall — call a function when a buffer becomes available, 9F-234
qtimeout — execute a function after a specified length of time, 9F-240
qunbufcall — cancel a pending qbufcall request, 9F-241
quntimeout — cancel previous timeout function call, 9F-242
qwait — STREAMS wait routines, 9F-243
qwait_sig — STREAMS wait routines, 9F-243
qwriter — asynchronous STREAMS perimeter upgrade, 9F-245

R

raw buffer
free a raw buffer header — freerbuf, 9F-175
get a raw buffer header — getrbuf, 9F-183
RD — get pointer to the read queue, 9F-32
read from an I/O port — inb, 9F-185
inl, 9F-185
inw, 9F-185
repinsb, 9F-185
repinsd, 9F-185
repinsw, 9F-185
readers/writer lock functions
— rw_destroy, 9F-256
— rw_downgrade, 9F-256
— rw_enter, 9F-256
— rw_exit, 9F-256
— rw_init, 9F-256
— rw_read_locked, 9F-256
— rw_tryenter, 9F-256
— rw_tryupgrade, 9F-256
— rwlock, 9F-256
remove a kstat from the system — kstat_delete, 9F-193
repinsb — read from an I/O port, 9F-185
repinsd — read from an I/O port, 9F-185
repinsw — read from an I/O port, 9F-185
repoutsb — write to an I/O port, 9F-215
repoutsd — write to an I/O port, 9F-215
repoutsw — write to an I/O port, 9F-215
resource map
allocate resource maps — rmallocmap, 9F-250
free resource maps — rmallocmap, 9F-250
retrieve kernel state information — drv_getparm,

9F-156
return index matching capability string — scsi_hba_lookup_capstr, 9F-273
return the larger of two integers — max, 9F-202
return the lesser of two integers — min, 9F-203
rmalloc — allocate space from a resource map, 9F-246
rmalloc_wait — allocate space from a resource map, 9F-249
rmfree — free space back into a resource map, 9F-251

S

SAMESTR — test if next queue is in the same stream, 9F-33
SCSI commands, make packet
— makecom, 9F-199
— makecom_g0, 9F-199
— makecom_g0_s, 9F-199
— makecom_g1, 9F-199
— makecom_g5, 9F-199
SCSI dma utility routines
— scsi_dmafree, 9F-264
— scsi_dmaget, 9F-264
SCSI HBA attach and detach routines — scsi_hba_attach, 9F-270
scsi_hba_detach, 9F-270
SCSI Host Bus Adapter system initialization and completion routines — scsi_hba_init, 9F-272
scsi_hba_fini, 9F-272
SCSI packet
allocate a SCSI packet in iopb map — get_pktiopb, 9F-177
free a packet in iopb map — free_pktiopb, 9F-177
free an allocated SCSI packet and its DMA resource — scsi_destroy_pkt, 9F-263
SCSI packet utility routines
— scsi_pktalloc, 9F-285
— scsi_pktfree, 9F-285
— scsi_resalloc, 9F-285
SCSI packet utility routines, *continued*
— scsi_resfree, 9F-285
scsi_abort — abort a SCSI command, 9F-258
scsi_alloc_consistent_buf — scsi dma utility for allocating an I/O buffer for SCSI DMA, 9F-259
scsi_cname — decode SCSI commands, 9F-261
scsi_destroy_pkt — free an allocated SCSI packet and its DMA resource, 9F-263
scsi_dname — decode SCSI peripheral device type, 9F-261
scsi_errmsg — display a SCSI request sense message, 9F-266
scsi_free_consistent_buf — free a previously allocated SCSI DMA I/O buffer, 9F-269
scsi_hba_attach — SCSI HBA attach and detach routines, 9F-270
scsi_hba_detach — SCSI HBA attach and detach routines, 9F-270
scsi_hba_fini — SCSI Host Bus Adapter system completion routines, 9F-272
scsi_hba_init — SCSI Host Bus Adapter system initialization routines, 9F-272
scsi_hba_lookup_capstr — return index matching capability string, 9F-273
scsi_hba_pkt_alloc — allocate and free a scsi_pkt structure, 9F-274
scsi_hba_pkt_free — allocate and free a scsi_pkt structure, 9F-274
scsi_hba_probe — default SCSI HBA probe function, 9F-276
scsi_hba_tran_alloc — allocate and free transport structures, 9F-277
scsi_hba_tran_free — allocate and free transport structures, 9F-277
scsi_ifgetcap — get SCSI transport capability, 9F-278
scsi_ifsetcap — set SCSI transport capability, 9F-278
scsi_init_pkt — prepare a complete SCSI packet, 9F-280
scsi_log — display a SCSI-device-related mes-

sage, 9F-283
scsi_mname — decode SCSI messages, 9F-261
scsi_poll — run a polled SCSI command on
behalf of a target driver, 9F-287
scsi_probe — utility for probing a scsi device,
9F-288
scsi_reset — reset a SCSI bus or target, 9F-290
scsi_rname — decode SCSI packet completion
reasons, 9F-261
scsi_slave — utility for SCSI target drivers to
establish the presence of a target, 9F-291
scsi_sname — decode SCSI sense keys, 9F-261
scsi_sync_pkt — synchronize CPU and I/O
views of memory, 9F-292
scsi_transport — request by a target driver to
start a SCSI command, 9F-293
scsi_unprobe — free resources allocated during
initial probing, 9F-294
scsi_unslave — free resources allocated during
initial probing, 9F-294
semaphore functions
— sema_destroy, 9F-295
— sema_init, 9F-295
— sema_p, 9F-295
— sema_p_sig, 9F-295
— sema_tryp, 9F-295
— sema_v, 9F-295
— semaphore, 9F-295
send a signal to a process — proc_signal, 9F-219
proc_ref, 9F-219
proc_unref, 9F-219
single direction elevator seek sort for buffers —
disksort, 9F-155
size in bytes
convert size in pages — ptob, 9F-221
convert to size in memory pages (round down)
— btop, 9F-51
convert to size in memory pages (round up) —
btopr, 9F-52
sprintf — format characters in memory, 9F-297
STREAMS message blocks
allocate — allocb, 9F-36
attach a user-supplied data buffer in place —
esballoc,
STREAMS message blocks, *continued*
9F-167
call a function when a buffer becomes available
— bufcall, 9F-314, 9F-53, 9F-234,
9F-241
call function when buffer is available — esbb-
call, 9F-169
concatenate bytes in a message — msgpullup,
9F-206, 9F-222
concatenate two — linkb, 9F-198
copy — copyb, 9F-66
erase the contents of a buffer — clrbuf, 9F-59
free all message blocks in a message —
freemsg, 9F-174
free one — freeb, 9F-173
remove from head of message — unlinkb,
9F-315
remove one form a message — rmvb, 9F-252
STREAMS message queue
insert a message into a queue — insq, 9F-186
STREAMS message queues, 9F-38
STREAMS Message queues
get next message — getq, 9F-182
STREAMS message queues
reschedule a queue for service — enableok,
9F-166
test for room — canput, 9F-57, 9F-58
STREAMS messages
copy a message — copymsg, 9F-70
flush for specified priority band — flush-
band, 9F-170
remove form queue — flushq, 9F-171, 9F-254
return the number of bytes in a message —
msgdsize, 9F-205
submit messages to the log driver — strlog,
9F-304
test whether a message is a data message —
datamsg, 9F-74
trim bytes — adjmsg, 9F-35
STREAMS multithreading
— mt-streams, 9F-207
qbufcall — call a function when a buffer
becomes available, 9F-234
qtimeout — execute a function after a

specified
STREAMS multithreading, *continued*
length of time, 9F-240
qunbufcall — cancel a pending qbufcall request, 9F-241
quntimeout — cancel previous timeout function call, 9F-242
qwait, qwait_sig — STREAMS wait routines, 9F-243
qwriter — asynchronous STREAMS perimeter upgrade, 9F-245
STREAMS put and service procedures
disable — qprocsoff, 9F-236
enable — qprocson, 9F-236
STREAMS queues
change information about a queue or band of the queue — strqset, 9F-306
enable a queue — qenable, 9F-235
get information about a queue or band of the queue — strqget, 9F-305
get pointer to queue's partner queue — OTHERQ, 9F-31
get pointer to the read queue — RD, 9F-32
number of messages on a queue — qsize, 9F-239
place a message at the head of a queue — putbq, 9F-225
prevent a queue from being scheduled — noenable, 9F-213
put a message on a queue — putq, 9F-233
send a control message to a queue — putctl, 9F-226, 9F-230
send a control message with a one-byte parameter to a queue — putctl1, 9F-228, 9F-232
send a message on a stream in the reverse direction — qreply, 9F-237
send a message to the next queue — putnext, 9F-229
test for flow control in specified priority band — bcanput, 9F-39
test if next queue is in the same stream — SAMESTR, 9F-33
STREAMS wait routines — qwait, qwait_sig, 9F-243
STREAMS write queues
get pointer for this module or driver — WR, 9F-34
swab — swap bytes in 16-bit halfwords, 9F-307
synchronize CPU and I/O views of memory — ddi_dma_sync, 9F-108, 9F-292
system DMA engine functions — ddi_dmae, 9F-110
ddi_dmae_1stparty, 9F-110
ddi_dmae_alloc, 9F-110
ddi_dmae_disable, 9F-110
ddi_dmae_enable, 9F-110
ddi_dmae_getcnt, 9F-110
ddi_dmae_getlim, 9F-110
ddi_dmae_prog, 9F-110
ddi_dmae_release, 9F-110
ddi_dmae_stop, 9F-110

T

testb — check for an available buffer, 9F-308
timeout — execute a function after a specified length of time, 9F-310
cancel previous timeout function call — untimeout, 9F-316

U

uio structure
add character — ureadc, 9F-318
remove a character — uwritec, 9F-319
uiomove — copy kernel data using uio structure, 9F-313
unfreezestr — freeze, thaw the state of a stream, 9F-176
update I/O kstat statistics
— kstat_queue, 9F-196
— kstat_runq_back_to_waitq, 9F-196
— kstat_runq_enter, 9F-196
— kstat_runq_exit, 9F-196
— kstat_waitq_enter, 9F-196
— kstat_waitq_exit, 9F-196
— kstat_waitq_to_runq, 9F-196

V

`vsprintf` — format characters in memory, 9F-320

W

write to an I/O port — `outb`, 9F-215
outl, 9F-215
outw, 9F-215
repoutsb, 9F-215
repoutsd, 9F-215
repoutsw, 9F-215