IRIX™ Device Driver Reference Pages

Document Number 007-2183-003

CONTRIBUTORS

Written by Susan Ellis
Production by Gloria Ackley

IRIX™ Device Driver Reference Pages
Document Number 007-2183-003

Contents

Introduction xv
About This Guide xv
Audience xv
Reference Material xvi
Notation and Syntax Conventions xvi
Document Overview xvii
Functions, Data Structures, and Defines xvii
Driver Data Definitions (D1) xviii
Driver Entry Points and Memory Mapping Extensions (D2 and D2X) xviii
Kernel Utilities and Utility Extensions (D3 and D3X) xx
Kernel Data Structures and Extensions (D4 and D4X) xxx
Kernel Definitions (D5) xxxi

Driver Data Definitions (D1) 1
intro(D1) 3
devflag(D1) 4
info(D1) 5
prefix(D1) 6

Driver Entry Points and Memory Mapping Extensions (D2 and D2X) 7
intro(D2) 9
close(D2) 10
halt(D2) 13
init(D2) 14
intr(D2) 16
ioctl(D2) 18

map(D2X) 20
mmap(D2) 22
open(D2) 24
poll(D2) 28
print(D2) 30
put(D2) 31
read(D2) 34
size(D2) 35
srv(D2) 36
start(D2) 39
strategy(D2) 40
unload(D2) 42
unmap(D2X) 43
write(D2) 44

Kernel Utilities and Utility Extensions (D3 and D3X) 47
intro(D3) 49
adjmsg(D3) 50
allocb(D3) 51
ASSERT(D3) 53
badaddr(D3X) 54
bcanput(D3) 55
bcopy(D3) 56
biodone(D3) 58
bioerror(D3) 60
biowait(D3) 61
bptophys(D3X) 62
bp_mapin(D3) 63
bp_mapout(D3) 64
brelse(D3) 65
btod(D3X) 66
btop(D3) 67
btopr(D3) 68

bufcall(D3) 69
bzero(D3) 72
canput(D3) 73
clrbuf(D3) 74
cmn_err(D3) 75
copyb(D3) 78
copyin(D3) 80
copymsg(D3) 82
copyout(D3) 84
cpsema(D3X) 86
cvsema(D3X) 87
datamsg(D3) 88
delay(D3) 90
dki_dcache_inval(D3X) 91
dki_dcache_wb(D3X) 92
dki_dcache_wbinval(D3X) 93
dma_map(D3X) 94
dma_mapaddr(D3X) 95
dma_mapalloc(D3X) 96
dma_mapfree(D3X) 97
drv_getparm(D3) 98
drv_hztousec(D3) 100
drv_priv(D3) 101
drv_setparm(D3) 102
drv_usectohz(D3) 104
drv_usecwait(D3) 105
dtimeout(D3) 106
dupb(D3) 108
dupmsg(D3) 110
eisa_dma_disable(D3X) 111
eisa_dma_enable(D3X) 112
eisa_dma_free_buf(D3X) 113
eisa_dma_free_cb(D3X) 114

eisa_dma_get_buf(D3X) 115
eisa_dma_get_cb(D3X) 116
eisa_dma_prog(D3X) 117
eisa_dma_stop(D3X) 118
eisa_dma_swstart(D3X) 119
enableok(D3) 120
esballoc(D3) 121
esbbcall(D3) 123
etoimajor(D3) 124
flushband(D3) 125
flushbus(D3X) 126
flushq(D3) 127
freeb(D3) 128
freemsg(D3) 129
freerbuf(D3) 130
freesema(D3X) 131
fubyte(D3X) 132
fuword(D3X) 133
geteblk(D3) 134
getemajor(D3) 136
geteminor(D3) 137
geterror(D3) 138
getmajor(D3) 139
getminor(D3) 140
getnextpg(D3X) 141
getq(D3) 142
getrbuf(D3) 143
hwcpin(D3X) 145
hwcpout(D3X) 146
initnsema(D3X) 147
initnsema_mutex(D3X) 148
insq(D3) 149
itimeout(D3) 151

itoemajor(D3) 153
kern_calloc(D3X) 154
kern_free(D3X) 155
kern_malloc(D3X) 156
kmem_alloc(D3) 157
kmem_free(D3) 159
kmem_zalloc(D3) 160
kvtophys(D3X) 162
linkb(D3) 163
LOCK(D3) 164
LOCK_ALLOC(D3) 166
LOCK_DEALLOC(D3) 168
makedevice(D3) 169
max(D3) 171
min(D3) 172
msgdsize(D3) 173
msgpullup(D3) 174
ngeteblk(D3) 175
noenable(D3) 177
OTHERQ(D3) 178
pcmsg(D3) 180
phalloc(D3) 182
phfree(D3) 183
physiock(D3) 184
pio_andb_rmw(D3X) 186
pio_andh_rmw(D3X) 187
pio_andw_rmw(D3X) 188
pio_badaddr(D3X) 189
pio_bcopyin(D3X) 190
pio_bcopyout(D3X) 191
pio_mapaddr(D3X) 192
pio_mapalloc(D3X) 193
pio_mapfree(D3X) 194

pio_orb_rmw(D3X) 195
pio_orh_rmw(D3X) 196
pio_orw_rmw(D3X) 197
pio_wbadaddr(D3X) 198
pollwakeup(D3) 199
pptophys(D3X) 200
proc_ref(D3) 201
proc_signal(D3) 202
proc_unref(D3) 203
psema(D3X) 204
ptob(D3) 205
putbq(D3) 206
putctl(D3) 207
putctl1(D3) 209
putnext(D3) 210
putq(D3) 211
qenable(D3) 212
qreply(D3) 213
qsize(D3) 214
RD(D3) 215
rmalloc(D3) 216
rmallocmap(D3) 219
rmalloc_wait(D3) 220
rmfree(D3) 221
rmfreemap(D3) 223
rmvb(D3) 224
rmvq(D3) 226
SAMESTR(D3) 227
scsi_alloc(D3X) 228
scsi_command(D3X) 229
scsi_free(D3X) 230
scsi_info(D3X) 231
sgset(D3X) 232

sleep(D3) 233
SLEEP_ALLOC(D3) 235
SLEEP_DEALLOC(D3) 236
SLEEP_LOCK(D3) 237
SLEEP_LOCKAVAIL(D3) 238
SLEEP_LOCK_SIG(D3) 239
SLEEP_TRYLOCK(D3) 241
SLEEP_UNLOCK(D3) 242
spl(D3) 243
strcat(D3) 245
strcpy(D3) 246
streams_interrupt(D3X) 247
STREAMS_TIMEOUT(D3X) 248
strlog(D3) 249
strqget(D3) 251
strqset(D3) 253
subyte(D3X) 254
suword(D3X) 255
TRYLOCK(D3) 256
uiomove(D3) 257
uiophysio(D3X) 259
unbufcall(D3) 261
undma(D3X) 263
unlinkb(D3) 264
UNLOCK(D3) 265
untimeout(D3) 266
untimeout_func(D3X) 267
ureadc(D3) 268
userdma(D3X) 270
uwritec(D3) 271
valusema(D3X) 273
vme_adapter(D3X) 274
vme_ivec_alloc(D3X) 275

vme_ivec_free(D3X) 276
vme_ivec_set(D3X) 277
volatile(D3X) 278
vpsema(D3X) 279
vsema(D3X) 280
v_getaddr(D3X) 281
v_gethandle(D3X) 282
v_getlen(D3X) 283
v_mapphys(D3X) 284
wakeup(D3) 285
wbadaddr(D3X) 286
WR(D3) 287

Kernel Data Structures and Extensions (D4 and D4X) 289
intro(D4) 291
buf(D4) 292
copyreq(D4) 296
copyresp(D4) 298
datab(D4) 299
eisa_dma_cb(D4X) 300
eisa_dma_buf(D4X) 304
free_rtn(D4) 306
iocblk(D4) 307
iovec(D4) 309
linkblk(D4) 310
module_info(D4) 311
msgb(D4) 313
qinit(D4) 315
queue(D4) 317
streamtab(D4) 319
stroptions(D4) 320
uio(D4) 323

Kernel Definitions (D5) 325
intro(D5) 327
errnos(D5) 328
messages(D5) 330
signals(D5) 332

Index 333

Tables

Table i Driver Data Definitions (D1) xviii
Table ii Driver Entry Points (D2) xix
Table iii Memory Mapping Extensions (D2X) xx
Table iv Kernel Utilities (D3) xx
Table v Utility Extensions (D3X) xxvi
Table vi Kernel Data Structures (D4) xxx
Table vii Data Structure Extensions (D4X) xxxi
Table viii Kernel Definitions (D5) xxxi

Introduction

About This Guide

This manual, the *IRIX™ Device Driver Reference Pages*, provides reference page (man page) information for developing UNIX® device drivers for IRIX 5.3 and later releases.

Audience

This manual is a guide to writing device drivers for Silicon Graphics® workstations and servers. It is intended for experienced C programmers who have a good working knowledge of UNIX internals and computer architecture.

Reference Material

For further references and background on writing device drivers, you may want to consult:

- *IRIX Device Driver Programming Guide*, Silicon Graphics, Inc., Part Number 007-0911-050
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988
- Kane, Gerry, and Joe Heinrich. *MIPS RISC ARCHITECTURE*, Prentice Hall, 1992
- Egan, Janet I., and Thomas J. Teixeira. *Writing a UNIX Device Driver.* John Wiley & Sons, 1992
- Hines, Robert M., and Spence Wilcox. *Device Driver Programming*, UNIX SVR4.2, UNIX Press, 1992
- *STREAMS Modules and Drivers*, UNIX SVR4.2, UNIX Press, 1992

Notation and Syntax Conventions

This guide uses the following notation and syntax conventions:

italic	In code it indicates arguments that you must replace with a valid value. In text, it is used to indicate commands, document titles, file names, and variables.
`courier bold`	Indicates functions, routines, and entry points names, for example **`read`**(D2).
`courier`	Indicates computer output and program listings.

Document Overview

This manual lists the functions, data structures, and kernel defines available for writing device drivers. It contains the following five sections:

- Driver Data Definitions (D1)
- Driver Entry Points and Memory Mapping Extensions (D2 and D2X)
- Kernel Utilities and Utility Extensions (D3 and D3X)
- Kernel Data Structures and Extensions (D4 and D4X)
- Kernel Definitions (D5)

Within each section, reference pages are arranged in alphabetical order except for the *intro* reference page, which appears first.

The remainder of this Introduction describes the reference pages included in each section.

Functions, Data Structures, and Defines

This section lists the functions and data structures that are used to develop a device driver.

Each driver is uniquely identified by a prefix string specified in its configuration file. The name of all the driver-supplied functions and global variables should begin with this prefix. This reduces the chance of a symbol collision with another driver. Any private functions defined by a driver that are not entry point functions should be declared as *static*. Also, any global variables that are private to the driver should be declared as *static*.

Driver Data Definitions (D1)

Table i lists the data definition functions used by a device driver.

Table i Driver Data Definitions (D1)

Function	Description
intro	Introduction to driver data
devflag	Driver flags
info	STREAMS driver and module information
prefix	Driver prefix

Driver Entry Points and Memory Mapping Extensions (D2 and D2X)

This section describes the “entry point functions” that provide the interfaces that the kernel needs from drivers. The kernel calls them when needed. Some are called at well-defined times, such as system start up and system shut down. Others are called as a result of I/O-related system calls or external events, such as interrupts from peripheral devices.

Each driver is organized into two logical parts:

- Base level – interacts with the kernel and the device on behalf of processes performing I/O operations.
- Interrupt level – interacts with the device and the kernel as a result of an event such as data arrival, and usually cannot be associated with any particular process. Each driver is uniquely identified by a prefix string specified in its configuration file.

Table ii is a list of the driver entry point functions.

Table ii Driver Entry Points (D2)

Function	Description
intro	Introduction to driver entry point routines
close	Relinquish access to a device
halt	Shut down the driver when the system shuts down
init	Initialize a device
intr	Process a device interrupts
ioctl	Control a character device
mmap	Support virtual mapping for memory-mapped device
open	Gain access to a device
poll	Poll entry point for a non-stream character driver
print	Display a driver message on the system console
put	Receive messages from the preceding queue
read	Read data from a device
size	Return size of logical block device
srv	Service queued messages
start	Initialize a device at system start-up
strategy	Perform block I/O
unload	Clean up a loadable kernel module
write	Write data to a device

Table iii lists the Silicon Graphics-specific memory mapping functions used by a device driver.

Table iii Memory Mapping Extensions (D2X)

Function	Description
map	Support virtual mapping for memory-mapped device
unmap	Support virtual unmapping for memory-mapped device

Kernel Utilities and Utility Extensions (D3 and D3X)

This section describes the kernel utility functions available for use by device drivers. Drivers must not call any kernel functions other than the ones described in this section. Unless otherwise stated, any kernel utility function that sleeps will do so such that signals will not interrupt the sleep.

Table iv is a list of the kernel utility functions.

Table iv Kernel Utilities (D3)

Function	Description
intro	Introduction to kernel utility routines
adjmsg	Trim bytes from a message
allocb	Allocate a message block
ASSERT	Verify assertion
bcanput	Test for flow control in a specified priority band
bcopy	Copy data between address locations in the kernel
biodone	Release buffer after block I/O and wakeup processes
bioerror	Manipulate error field within a buffer header
biowait	Suspend processes pending completion of block I/O
bp_mapin	Allocate virtual address space for buffer page list
bp_mapout	Deallocate virtual address space for buffer page list

Table iv (continued) Kernel Utilities (D3)

Function	Description
brelse	Return a buffer to the system's free list
btop	Convert size in bytes to size in pages (round down)
btopr	Convert size in bytes to size in pages (round up)
bufcall	Call a function when a buffer becomes available
bzero	Clear memory for a given number of bytes
canput	Test for room in a message queue
clrbuf	Erase the contents of a buffer
cmn_err	Display an error message or panic the system
copyb	Copy a message block
copyin	Copy data from a user buffer to a driver buffer
copymsg	Copy a message
copyout	Copy data from a driver buffer to a user buffer
datamsg	Test whether a message is a data message
delay	Delay process execution for a specified of clock ticks
drv_getparm	Retrieve kernel state information
dvr_hztousec	Convert clock ticks to microseconds
drv_priv	Determine whether credentials are privileged
drv_setparm	Set kernel state information
drv_usectohz	Convert microseconds to clock ticks
drv_usecwait	Busy-wait for specified interval
dtimeout	Execute a function on a specified processor, after a specified length of time
dupb	Duplicate a message block
dupmsg	Duplicate a message

Table iv (continued) Kernel Utilities (D3)

Function	Description
enableok	Allow a queue to be serviced
esballoc	Allocate a message block using an externally-supplied buffer
esbbcall	Call a function when an externally-supplied buffer can be allocated
etoimajor	Convert external to internal major device number
flushband	Flush messages in a specified priority band
flushq	Flush messages on a queue
freeb	Free a message block
freemsg	Free a message
freerbuf	Free a raw buffer header
geteblk	Get an empty buffer
getemajor	Get external major device number
geteminor	Get external minor device number
geterror	Retrieve error number from a buffer header
getmajor	Get internal major device number
getminor	Get internal minor device number
getq	Get the next message from a queue
getrbuf	Get a raw buffer header
insq	Insert a message into a queue
itimeout	Execute a function after a specified length of time
itoemajor	Convert internal to external major device number
kmem_alloc	Allocate space from kernel free memory
kmem_free	Free previously allocated kernel memory
kmem_zalloc	Allocate and clear space from kernel free memory

Table iv (continued) Kernel Utilities (D3)

Function	Description
linkb	Concatenate two message blocks
LOCK	Acquire a basic lock
LOCK_ALLOC	Allocate and initialize a basic lock
LOCK_DEALLOC	Deallocate an instance of a basic lock
makedevice	Make device number from major and minor numbers
max	Return the larger of two integers
min	Return the lesser of two integers
msgdsize	Return number of bytes of data in a message
msgpullup	Concatenate bytes in a message
ngeteblk	Get an empty buffer of the specified size
noenable	Prevent a queue from being scheduled
OTHERQ	Get a pointer to queue's partner queue
pcmsg	Test whether a message is a priority control message
phalloc	Allocate and initialized a pollhead structure
phfree	Free a pollhead structure
physiock	Validate and issue raw I/O request
pollwakeup	Inform polling process that an event has occurred
proc_ref	Obtain a reference to a process for signaling
proc_signal	Send a signal to a process
proc_unref	Release a reference to a process
ptob	Convert size in pages to size in bytes
putbq	Place a message at the head of a queue
putctl	Send a control message to a queue

Table iv (continued) Kernel Utilities (D3)

Function	Description
putctl1	Send a control message with a one-byte parameter to a queue
putnext	Send a message to the next queue
putq	Put a message on a queue
qenable	Schedule a queue's service routine to be run
qreply	Send a message in the opposite direction in a stream
qsize	Find the number of message on a queue
RD	Get a pointer to the read queue
rmalloc	Allocate space from a private space management map
rmallocmap	Allocate and initialize a private space management map
rmalloc_wait	Allocate space from a private space management map
rmfree	Free space into a private space management map
rmfreemap	Free private space management map
rmvb	Remove a message block from a message
rmvq	Remove a message from a queue
SAMESTR	Test if next queue is of the same type
sleep	Suspend process execution pending occurrence of an event
SLEEP_ALLOC	Allocate and initialize a sleep lock
SLEEP_DEALLOC	Deallocate an instance of a sleep lock
SLEEP_LOCK	Acquire a sleep lock
SLEEP_LOCKAVAIL	Query whether a sleep lock is available
SLEEP_LOCK_SIG	Acquire a sleep lock
SLEEP_TRYLOCK	Try to acquire a sleep lock
SLEEP_UNLOCK	Release a sleep lock

Table iv (continued) Kernel Utilities (D3)

Function	Description
spl	Block / allow interrupts on a processor
strcat	Concatenate strings
strcpy	Copy a string
strlog	Submit messages to the log driver
strqget	Get information about a queue or band of the queue
strqset	Change information about a queue or band of the queue
TRYLOCK	Try to acquire a basic lock
uiomove	Copy data using uio structure
unbufcall	Cancel a pending bufcall request
unlinkb	Remove a message block from the head of a message
UNLOCK	Release a basic lock
untimeout	Cancel previous timeout request
ureadc	Copy a character to space described by **`uio`** structure
uwritec	Return a character from space described by **`uio`** structure
wakeup	Resume suspended process execution
WR	Get a pointer to the write queue

Table v lists utility functions that are specific to Silicon Graphics. These functions are probably different than functions of the same name in another device driver reference page book, such as the UNIX "red book," so read them carefully.

Table v Utility Extensions (D3X)

Function	Description
badaddr	Check for bus error when reading an address
bptophys	Get physical address of buffer data
btod	Convert from bytes to disk sectors
cpsema	Conditionally perform a "P" or wait semaphore operation
cvsema	Conditionally perform a "V" or wait semaphore operation
dki_dcache_inval	Invalidate the data cache for a given range of virtual addresses
dki_dcache_wb	Write back the data cache for a given range of virtual addresses
dki_dcache_wbinval	Write back and invalidate the data cache for a given range of virtual addresses
dma_map	Load DMA mapping registers for an imminent transfer
dma_mapaddr	Return the "bus virtual" address for a given map and address
dma_mapalloc	Allocate a DMA map
dma_mapfree	Free a DMA map
eisa_dma_disable	Disable recognition of hardware requests on a DMA channel
eisa_dma_enable	Enable recognition of hardware requests on a DMA channel
eisa_dma_free_buf	Free a previously allocated DMA buffer descriptor
eisa_dma_free_cb	Free a previously allocated DMA command block
eisa_dma_get_buf	Allocated DMA buffer descriptor

Table v (continued) Utility Extensions (D3X)

Function	Description
eisa_dma_get_cb	Allocated a DMA command block
eisa_dma_prog	Program a DMA operation for a subsequent software request
eisa_dma_stop	Stop software-initiated DMA operation on a channel and release it
eisa_dma_swstart	Initiate a DMA operation via software request
flushbus	Make sure contents of the write buffer are flushed to the system bus
freesema	Free the resources associated with a semaphore
fubyte	Fetch (read) a byte from user space
fuword	Fetch (read) a word from user space
getnextpg	Get next page pointer
hwcpin	Copy data from device memory to main memory using 16-bit reads
hwcpout	Copy data from main memory to device memory using 16-bit writes
initnsema	Allocate a semaphore and initialize it to a given value
initnsema_mutex	Initialize a mutex semaphore to one
kern_calloc	Allocate storage for objects of a specified size
kern_free	Free kernel memory space
kern_malloc	Allocate kernel virtual memory
kvtophys	Get physical address of buffer data
pio_andb_rmw	Byte VME-bus read-modify-write cycle routines
pio_andh_rmw	Half-word VME-bus read-modify-write cycle routine
pio_andw_rmw	Word VME-bus read-modify-write cycle routines
pio_badaddr	Check for bus error when reading an address

Table v (continued) Utility Extensions (D3X)

Function	Description
pio_bcopyin	Copy data from VME bus address to kernel's virtual space
pio_bcopyout	Copy data from kernel's virtual space to VME bus address
pio_mapaddr	Used with FIXED maps to generate a kernel pointer to VME bus space
pio_mapalloc	Allocate a PIO map
pio_mapfree	Free up a previously allocated PIO map
pio_orb_rmw	VME-bus read-modify-write cycle routines
pio_orh_rmw	VME-bus read-modify-write cycle routines
pio_orw_rmw	VME-bus read-modify-write cycle routines
pio_wbadaddr	Check for bus error when writing to an address
pptophys	Convert page pointer to physical address
psema	Perform a "P" or wait semaphore operation
scsi_alloc	Allocate communication channel between host adapter driver and a kernel level SCSI device driver
scsi_command	Issue a command to a SCSI device
scsi_free	Free communication channel between host adapter driver and a kernel level SCSI device driver
scsi_info	Get information about a SCSI device
sgset	Assign physical addresses to a vector of software scatter-gather registers
streams_interrupt	Synchronize interrupt-level function with STREAMS mechanism
STREAMS_TIMEOUT	Synchronize timeout with STREAMS mechanism
subyte	Set (write) a byte to user space
suword	Set (write) a word to user space
uiophysio	Set up user data space for I/O

Table v (continued) Utility Extensions (D3X)

Function	Description
undma	Unlock physical memory in user space
untimeout_func	Cancel a previous invocation of timeout by function
userdma	Lock, unlock physical memory in user space
valusema	Return the value associated with a semaphore
vme_adapter	Determine VME adapter
vme_ivec_alloc	Allocate a VME bus interrupt VECTOR
vme_ivec_free	Free up a VME bus interrupt VECTOR
vme_ivec_set	Register a VME bus interrupt handler
volatile	Inform the compiler of volatile variables
vpsema	Perform an atomic “V” and “P” semaphore operation on two semaphores
vsema	Perform a “V” or signal semaphore operation
v_getaddr	Get the user address associated with virtual handle
v_gethandle	Get unique identifier associated with virtual handle
v_getlen	Get length of user address space associated with virtual handle
v_mapphys	Map physical addresses into user address space
wbadaddr	Check for bus error when writing to an address

Kernel Data Structures and Extensions (D4 and D4X)

This section describes the kernel data structures a developer might need to use in a device driver. Driver developers should not declare arrays of these structures, as the size of any structure might change between releases. Two exceptions to this are the **iovec**(D4) and **uio**(D4) structures.

Drivers can only reference those structure members described on the reference page. The actual data structures may have additional structure members beyond those described, but drivers must not reference them.

Some structure members are flags fields that consist of a bitmask of flags. Drivers must never directly assign values to these structure members. Drivers should only set and clear flags they are interested in, since the actual implementation may contain unlisted flags.

Data structures that are “black boxes” to drivers are not described in this section. These structures are referenced on the reference pages where they are used. Drivers should not be written to use any of their structure members. Their only valid use is passing pointers to the structures to the particular kernel functions.

Table vi is a list of the Silicon Graphics kernel data structures.

Table vi Kernel Data Structures (D4)

Function	Description
intro	Introduction to kernel data structures
buf	Block I/O data transfer structure
copyreq	STREAMS transparent **ioctl** copy request structure
copyresp	STREAMS transparent **ioctl** copy response structure
datab	STREAMS data block structure
free_rtn	STREAMS driver’s message free routine structure
iocblk	STREAMS **ioctl** structure
iovec	data storage structure for I/O using **uio**
linkblk	STREAMS multiplexor link structure

Table vi (continued) Kernel Data Structures (D4)

Function	Description
module_info	STREAMS driver and module information structure
msgb	STREAMS message block structure
qinit	STREAMS queue initialization structure
queue	STREAMS queue structure
streamstab	STREAMS driver and module declaration structure
stroptions	STREAMS head option structure
uio	Scatter / gather I / O request structure

Table vii is a list of data structure extensions supported by Silicon Graphics.

Table vii Data Structure Extensions (D4X)

Function	Description
eisa_dma_buf	EISA DMA buffer descriptor structure
eisa_dma_cb	DMA command block structure

Kernel Definitions (D5)

Table viii is a list of kernel defines supported by Silicon Graphics.

Table viii Kernel Definitions (D5)

Function	Description
intro	Introduction to kernel #define's
errnos	Error numbers
messages	STREAMS messages
signals	Signal numbers

Driver Data Definitions (D1)

NAME

intro – introduction to driver data

SYNOPSIS

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

DESCRIPTION

This section describes the data definitions a developer needs to include in a device driver. The system finds this information in an implementation-specific manner, usually tied to the way system configuration is handled.

USAGE

Each driver is uniquely identified by a prefix string specified in its configuration file. The name of all the driver-supplied routines and global variables should begin with this prefix. This will reduce the chance of a symbol collision with another driver. Any private routines defined by a driver that are not entry point routines should be declared as **static**. Also, any global variables that are private to the driver should be declared as **static**.

NAME

devflag – driver flags

SYNOPSIS

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

int  prefixdevflag = 0;
```

DESCRIPTION

Every driver must define a global integer variable called *prefix***devflag**. This variable contains a bitmask of flags used to specify the driver's characteristics to the system.

The valid flags that may be set in *prefix***devflag** are:

D_MP	The driver is multithreaded (it handles its own locking and serialization).
D_WBACK	Writes back cache before strategy routine.
D_OLD	The driver uses the old-style interface (pre-5.0 release).

If no flags are set for the driver, then *prefix***devflag** should be set to 0. If this is not done, then **lboot** will assume that this is an old style drive, and it will set **D_OLD** flag as a default.

REFERENCES

physiock(D3)

NAME

info – STREAMS driver and module information

SYNOPSIS

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

struct streamtab prefixinfo = { . . . };
```

DESCRIPTION

Every STREAMS driver and module must define a global **streamtab**(D4) structure so that the system can identify the entry points and interface parameters.

REFERENCES

streamtab(D4)

NAME

prefix – driver prefix

SYNOPSIS

int *prefix***close();**
int *prefix***open();**
. . .

DESCRIPTION

Every driver must define a unique prefix. This enables driver entry points to be identified by configuration software and decreases the possibility of global symbol collisions in the kernel.

USAGE

The prefix is usually specified in a configuration file. The maximum length of the prefix is implementation-defined. Driver entry points names are created by concatenating the driver prefix with the name for the entry point.

Examples

An ETHERNET driver might use a driver prefix of ‘‘**en**.’’ It would define the following entry points: **enclose**, **eninit**, **enintr**, **enopen**, **enwput**, **enrsrv**, and **enwsrv**. It would also define the data symbols **endevflag** and **eninfo**.

REFERENCES

devflag(D1), **info**(D1), **chpoll**(D2), **close**(D2), **halt**(D2), **init**(D2), **intr**(D2), **ioctl**(D2), **mmap**(D2), **open**(D2), **print**(D2), **put**(D2), **read**(D2), **size**(D2), **srv**(D2), **start**(D2), **strategy**(D2), **unload**(D2), **write**(D2)

Driver Entry Points and Memory Mapping Extensions (D2 and D2X)

NAME

`intro` – introduction to driver entry point routines

SYNOPSIS

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

DESCRIPTION

This section describes the routines a developer needs to include in a device driver or STREAMS module.

USAGE

The routines described in this section are called ‘‘entry point routines’’ because they provide the interfaces that the kernel needs from drivers and STREAMS modules. The kernel calls these routines when needed. Some are called at well-defined times, such as system start up and system shut down. Others are called as a result of I/O-related system calls or external events, such as interrupts from peripheral devices.

Each driver or module is organized into two logical parts: the base level and the interrupt level. The base level interacts with the kernel and the device on behalf of processes performing I/O operations. The interrupt level interacts with the device and the kernel as a result of an event such as data arrival, and usually cannot be associated with any particular process.

Each driver or module is uniquely identified by a prefix string specified in its configuration file. The name of all the driver-supplied routines and global variables should begin with this prefix. This will reduce the chance of a symbol collision with another driver or module. Any private routines defined by a driver or module that are not entry point routines should be declared as **`static`**. Also, any global variables that are private to the driver or module should be declared as **`static`**.

In general, any number of instances of the same driver (or module) entry point routine can be running concurrently. It is the responsibility of the driver or module to synchronize access to its private data structures.

NAME

close – relinquish access to a device

SYNOPSIS

Block and Character Synopsis

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
```

int *prefix***close(dev_t** *dev*, **int** *flag*, **int** *otyp*, **cred_t ****crp***);**

Block and Character Arguments

dev	Device number.
flag	File status flags.
otyp	Parameter supplied so that the driver can determine how many times a device was opened and for what reasons.
crp	Pointer to the user credential structure.

STREAMS Synopsis

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>
```

int *prefix***close(queue_t ****q*, **int** *flag*, **cred_t ****crp***);**

STREAMS Arguments

q	Pointer to queue used to reference the read side of the driver.
flag	File status flag.
crp	Pointer to the user credential structure.

DESCRIPTION

Block and Character Description

The **close** routine ends the connection between the user process and the device, and prepares the device (hardware and software) so that it is ready to be opened again.

Valid values for *flag* and their definitions can be found in **open**(D2).

The values for *otyp* are mutually exclusive:

OTYP_BLK	Close was through the block interface for the device.
OTYP_CHR	Close was through the raw / character interface for the device.
OTYP_LYR	Close a layered device. This flag is used when one driver calls another driver's **close** routine.

For **OTYP_BLK** and **OTYP_CHR**, a device may be opened simultaneously by multiple processes and the driver **open** routine is called for each open, but the kernel will only call the **close** routine when the last process using the device issues a **close**(2) system call or exits.

There is one exception to this rule. If a device is opened through both its character and its block interfaces, then there will be one close per interface. For example, if the same device is opened twice through its block interface and three times through its character interface, then there will be two calls to the driver's close routine; one when the block interface is finished being used, and one when the character interface is finished being used.

For **OTYP_LYR**, there will be one such close for every corresponding open. Here, the driver should count each open and close based on the *otyp* parameter to determine when the device should really be closed.

STREAMS Description

The **close** routines of STREAMS drivers and modules are called when a stream is dismantled or a module popped. The steps for dismantling a stream are performed in the following order. First, any non-persistent multiplexor links present are unlinked and the lower streams are closed. Next, the following steps are performed for each module or driver on the stream, starting at the head and working toward the tail:

1. The write queue is given a chance to drain.
2. Interrupts from STREAMS devices are blocked.
3. The **close** routine is called.
4. The module or driver is removed from the stream.
5. Any remaining messages on the queues are freed.

Return Values

The **close** routine should return 0 for success, or the appropriate error number. Refer to **errnos**(D5) for a list of DDI / DKI error numbers. Return errors rarely occur, but if a failure is detected, the driver should still close the device and then decide whether the severity of the problem warrants displaying a message on the console.

USAGE

This entry point is required in all drivers and STREAMS modules.

A **close** routine could perform any of the following general functions, depending on the type of device and the service provided:

disable device interrupts

hang up phone lines

rewind a tape

deallocate buffers from a private buffering scheme

unlock an unsharable device (that was locked in the **open** routine)

flush buffers

notify a device of the close

cancel any pending timeout or bufcall routines that access data that are deinitialized or deallocated during close

deallocate any resources allocated on open

Synchronization Constraints

The **close** routine has user context and can sleep. However, STREAMS drivers and modules must sleep such that signals do not cause the sleep to longjump [see **sleep**(D3)]. Also, if a **close** routine does sleep, it is important that the driver writer synchronize the driver's **open** and **close** routines, since a driver can be reopened while being closed.

If the **FNDELAY** or **FNONBLOCK** flags are specified in the *flag* argument, the driver should try to avoid sleeping, if possible, during close processing.

REFERENCES

drv_priv(D3), **errnos**(D5), **open**(D2), **queue**(D4), **unbufcall**(D3), **untimeout**(D3), **sleep**(D3)

NAME

`halt` – shut down the driver when the system shuts down

SYNOPSIS

`void` *prefix***`halt(void);`**

DESCRIPTION

The **`halt`** routine if present, is called to shut down the driver when the system is shut down. After the **`halt`** routine is called, no more calls will be made to the driver entry points.

Return Values

None

USAGE

This entry point is optional.

The device driver shouldn't assume that the interrupts are enabled. The driver should make sure that no interrupts are pending from its device, and inform the device that no more interrupts should be generated.

Synchronization Constraints

User context is not available, so the driver's **`halt`** routine should not sleep.

NAME

init – initialize a device

SYNOPSIS

void *prefix***init(void);**

DESCRIPTION

The **init** routine executes during system initialization to initialize drivers and the devices they control.

Return Values

None

USAGE

This entry point is optional.

Although **init** and **start** routines both perform initialization tasks, they execute at different times during system start-up. For this reason, they should be used to handle different types of initialization tasks.

init routines:

execute during system initialization

handle any driver and device setup and initialization that must take place before system services are initialized (for example, perform any setup and initialization that must be done before device interrupts are enabled)

may only call the kernel functions listed below

start routines:

execute after system services are initialized

handle all driver and device setup and initialization that can take place after system services are initialized (most driver setup and initialization tasks can be performed at this time, using a **start** routine)

handle any driver and device setup and initialization that can only take place after system services are initialized (for example, perform any setup and initialization that must be done after device interrupts are enabled)

Types of activities performed by the **init** routine include initializing data structures, allocating memory for private data, mapping the device into virtual address space, and initializing hardware.

The following kernel functions can be called from the driver's **`init`** routine:

`ASSERT`	**`eisa_dma_get_buf`**	**`getmajor`**
`bcmp`	**`eisa_dma_get_cb`**	**`getminor`**
`bcopy`	**`eisa_dma_prog`**	**`itoemajor`**
`bioreset`	**`eisa_dma_stop`**	**`kmem_alloc`**
`btop`	**`eisa_dma_swstart`**	**`kmem_free`**
`btopr`	**`drv_getparm`**	**`kmem_zalloc`**
`bzero`	**`drv_hztousec`**	**`makedevice`**
`cmn_err`	**`drv_usectohz`**	**`max`**
`eisa_dma_disable`	**`drv_usecwait`**	**`min`**
`eisa_dma_enable`	**`etoimajor`**	**`rmalloc`**
`eisa_dma_free_buf`	**`getemajor`**	**`rminit`**
`eisa_dma_free_cb`	**`geteminor`**	**`rmfree`**

On multiprocessor systems, the following additional kernel functions can be called from the driver's **`init`** routine:

`LOCK_ALLOC`	**`phfree`**	**`rmfreemap`**
`phalloc`	**`rmallocmap`**	**`SLEEP_ALLOC`**

`init` routines for dynamically loadable modules are not called during system start-up as they are for statically linked modules. A loadable module's initialization is called each time the module is loaded into a running system.

Synchronization Constraints

Functions that can result in the caller sleeping, or that require user context, such as **`sleep`**(D3), may not be called from **`init`**. Any function that provides a flag to prevent it from sleeping must be called such that the function does not sleep.

REFERENCES

`start`(D2)

intr(D2)

NAME

`intr` – process a device interrupt

SYNOPSIS

`void` *prefix***`intr(int`** *ivec***`);`**

Arguments

ivec	Number used by the operating system to associate a driver's interrupt handler with an interrupting device. For a VME device, this number can be the logical device number, the interrupt vector number, or the address which is set by **`vme_ivec_set`**(). For a SCSI device, this number is a pointer to a **`scsi_request_t`** type structure.

DESCRIPTION

The **`intr`** routine is the interrupt handler for both block and character hardware drivers, as well as for non-driver hardware modules.

Return Values

None

USAGE

This entry point is only required for those modules that interface to hardware that interrupts the host computer. It is not used with software drivers.

The interrupt handler is responsible for determining the reason for an interrupt, servicing the interrupt, and waking up any base-level driver processes sleeping on any events associated with the interrupt.

For example, when a disk drive has transferred information to the host to satisfy a read request, the disk drive's controller generates an interrupt. The CPU acknowledges the interrupt and calls the interrupt handler associated with that controller and disk drive. The interrupt routine services the interrupt and then wakes up the driver base-level process waiting for data. The base-level portion of the driver then conveys the data to the user.

In general, most interrupt routines do the following tasks:

keep a record of interrupt occurrences

return immediately if no devices controlled by a driver caused the interrupt (only for systems supporting shared interrupts)

interpret the interrupt routine argument *ivec*

reject requests for devices that are not served by the device's controller

process interrupts that happen without cause (called spurious interrupts)

handle all possible device errors

wake processes that are sleeping on any events associated with the interrupt

There are also many tasks the **intr** routine must perform that are driver-type and device specific. For example, the following types of drivers require different functions from their **intr** routines:

A block driver dequeues requests and wakes up processes sleeping on an I/O request.

A terminal driver receives and sends characters.

A printer driver ensures that characters are sent.

In addition, the functions of an **intr** routine are device dependent. You should know the exact chip set that produces the interrupt for your device. You need to know the exact bit patterns of the device's control and status register and how data is transmitted into and out of your computer. These specifics differ for every device you access.

The **intr** routine for an intelligent controller that does not use individual interrupt vectors for each subdevice must access the completion queue to determine which subdevice generated the interrupt. It must also update the status information, set/clear flags, set/clear error indicators, and so forth to complete the handling of a job. The code should also be able to handle a spurious completion interrupt identified by an empty completion queue. When the routine finishes, it should advance the unload pointer to the next entry in the completion queue.

If the driver called **biowait**(D3) or **sleep**(D3) to await the completion of an operation, the **intr** routine must call **biodone**(D3) or **wakeup**(D3) to signal the process to resume.

The interrupt routine runs at the processor level associated with the interrupt level for the given device. Lower priority interrupts are deferred while the interrupt routine is active. Certain processor levels can block different interrupts. See **spl**(D3) for more information.

uiomove(D3), **ureadc**(D3), and **uwritec**(D3) cannot be used in an interrupt routine when the **uio_segflg** member of the **uio**(D4) structure is set to **UIO_USERSPACE** (indicating a transfer between user and kernel space).

Synchronization Constraints

The **intr** routine must never:

use functions that sleep

drop the interrupt priority level below the level at which the interrupt routine was entered

call any function or routine that requires user context (that is, if it accesses or alters information associated with the running process)

REFERENCES

biodone(D3), **biowait**(D3), **spl**(D3), **wakeup**(D3), **vme_ivec_set**(D3X)

NAME

ioctl – control a character device

SYNOPSIS

```
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/ddi.h>

int prefixioctl(dev_t dev, int cmd, void *arg,
          int mode, cred_t *crp, int *rvalp);
```

Arguments

dev	Device number.
cmd	Command argument the driver **ioctl** routine interprets as the operation to be performed.
arg	Passes parameters between the user and the driver. The interpretation of the argument is dependent on the command and the driver. For example, the argument can be an integer, or it can be the address of a user structure containing driver or hardware settings.
mode	Contains the file modes set when the device was opened. The driver can use this to determine if the device was opened for reading (**FREAD**), writing (**FWRITE**), and so on. See **open**(D2) for a description of the values.
crp	Pointer to the user credential structure.
rvalp	Pointer to the return value for the calling process. The driver may elect to set the value if the **ioctl**(D2) succeeds.

DESCRIPTION

The **ioctl**(D2) routine provides non-STREAMS character drivers with an alternate entry point that can be used for almost any operation other than a simple transfer of data.

The **ioctl** routine is basically a **switch** statement, with each **case** definition corresponding to a different **ioctl** command identifying the action to be taken.

Return Values

The **ioctl** routine should return 0 on success, or the appropriate error number on failure. The system call will usually return 0 on success or –1 on failure. However, the driver can choose to have the system call return a different value on success by passing the value through the *rvalp* pointer.

USAGE

This entry point is optional, and is valid for character device drivers only.

Most often, **ioctl** is used to control device hardware parameters and establish the protocol used by the driver in processing data. I/O control commands are used to implement terminal settings, to format disk devices, to implement a trace driver for debugging, and to flush queues.

If the third argument, *arg*, is a pointer to user space, the driver can use **copyin**(D3) and **copyout**(D3) to transfer data between kernel and user space.

STREAMS drivers do not have **ioctl** routines. The stream head converts I/O control commands to **M_IOCTL** messages, which are handled by the driver's **put**(D2) or **srv**(D2) routine.

Synchronization Constraints

The **ioctl** routine has user context and can sleep.

Warnings

An attempt should be made to keep the values for driver-specific I/O control commands distinct from others in the system. Each driver's I/O control commands are unique, but it is possible for user-level code to access a driver with an I/O control command that is intended for another driver, which can have serious results.

A common method to assign I/O control command values that are less apt to be duplicated is to compose the commands from some component unique to the driver (such as a module name or ID), and a counter, as in:

```
#define PREFIX              ('h'<<16|'d'<<8)
#define COMMAND1        (PREFIX|1)
#define COMMAND2        (PREFIX|2)
#define COMMAND3        (PREFIX|3)
```

REFERENCES

copyin(D3), **copyout**(D3), **drv_priv**(D3), **errnos**(D5), **open**(D2)

NAME

map – support virtual mapping for memory-mapped device

SYNOPSIS

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

int prefixmap(dev_t dev, vhandl_t *vt, off_t off, int len, int prot);
```

Arguments

dev	Device whose memory is to be mapped.
vt	A pointer to the kernel-resident data structure that describes the virtual space to which the device memory will be mapped. Your driver needs this pointer when calling kernel service routines (i.e., **v_mapphys**(D3X)).
off	Offset within device memory at which mapping begins.
len	The length of the device memory to be mapped into the user's address space.
prot	Protection flags from *mman.h*.

DESCRIPTION

The map entry point provides a way to support drivers for memory-mapped devices. A memory-mapped device has memory that can be mapped into a process's address space. The **mmap**(2) system call allows this device memory to be mapped into user space for direct access by the user application (this way no kernel buffering or system call overhead is incurred).

Return Values

If the protection and offset are valid for the device, the driver should return 0. Otherwise, the appropriate error number should be returned.

USAGE

This entry point is optional, and valid for memory-mapped device drivers only.

Valid values for *prot* are:

PROT_READ
: Page can be read.

PROT_WRITE
: Page can be written.

PROT_EXEC
: Page can be executed.

PROT_ALL
: All of the above.

Your driver should treat *vt* as an opaque and should not try to directly set any of the member values. To map physical addresses into user address space, drivers should use the **v_mapphys**(D3X) function. Use **v_gethandle**(D3X) if your driver must remember several virtual handles.

Synchronization Constraints

The map routine has user context and can sleep.

REFERENCES

unmap(D2X), **v_getaddr**(D3X), **v_getlen**(D3X), **v_gethandle**(D3X), **v_mapphys**(D3X)

NAME

mmap – support virtual mapping for memory-mapped device

SYNOPSIS

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

int prefixmmap(dev_t dev, off_t off, int prot);
```

Arguments

dev	Device whose memory is to be mapped.
off	Offset within device memory at which mapping begins.
prot	Protection flags from **mman.h**.

DESCRIPTION

The **mmap** entry point provides a way to support character drivers for memory-mapped devices. A memory-mapped device has memory that can be mapped into a process's address space. The **mmap**(2) system call, when applied to a character special file, allows this device memory to be mapped into user space for direct access by the user application (this way no kernel buffering or system call overhead is incurred).

The **mmap** routine checks if the offset is within the range of pages supported by the device. For example, a device that has 32K bytes of memory that can be mapped into user space should not support offsets greater than, or equal to, 32K. If the offset does not exist, then **NOPAGE** is returned. If the offset does exist, the **mmap** routine returns the physical page ID for the page at offset *off* in the device's memory.

Return Values

If the protection and offset are valid for the device, the driver should return the physical page ID. Otherwise, **NOPAGE** should be returned.

USAGE

This entry point is optional, and valid for memory-mapped character device or character pseudo-device drivers only.

Valid values for *prot* are:

PROT_READ	Page can be read.
PROT_WRITE	Page can be written.
PROT_EXEC	Page can be executed.
PROT_ALL	All of the above.

Synchronization Constraints

The **mmap** routine has user context and can sleep.

REFERENCES

map(D2X), **unmap**(D2X)

NAME

open – gain access to a device

SYNOPSIS

Block and Character Synopsis

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
```

int *prefix***open(dev_t ****devp*, **int** *oflag*, **int** *otyp*, **cred_t ****crp***);**

Block and Character Arguments

devp	Pointer to a device number.
oflag	Information passed from the user that instructs the driver on how to open the file.
otyp	Parameter supplied so that the driver can determine how many times a device was opened and for what reasons.
crp	Pointer to the user credential structure.

STREAMS Synopsis

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stream.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>
```

int *prefix***open(queue_t ****q*, **dev_t ****devp*, **int** *oflag*, **int** *sflag*, **cred_t ****crp***);**

STREAMS Arguments

q	Pointer to the queue used to reference the read side of the driver.
devp	Pointer to a device number. For modules, *devp* always points to the device number associated with the driver at the end (tail) of the stream.
oflag	Open flags.
sflag	STREAMS flag.
crp	Pointer to the user credential structure.

Return Values

The **open** routine should return 0 for success, or the appropriate error number.

DESCRIPTION

Block and Character Description

The driver's **open** routine is called to prepare a device for further access. It is called by the kernel during an **open**(2) or a **mount**(2) of the device special file. For non-STREAMS drivers, it can also be called from another (layered) driver.

The bit settings for **oflag** are found in **file.h**. Valid settings are:

FEXCL	Interpreted in a driver-dependent manner. Some drivers interpret this flag to mean open the device with exclusive access (fail all other attempts to open the device.)
FNDELAY	Open the device and return immediately without sleeping (do not block the open even if there is a problem.)
FNONBLOCK	Open the device and return immediately without sleeping (do not block the open even if there is a problem.)
FREAD	Open the device with read access permission.
FWRITE	Open the device with write access permission.

Valid values for *otyp* are defined in **open.h**. The values are mutually exclusive:

OTYP_BLK	Open occurred through block interface for the device.
OTYP_CHR	Open occurred through the raw / character interface for the device.
OTYP_LYR	Open a layered device. This flag is used when one driver calls another driver's **open** routine.

STREAMS Description

The STREAMS module **open** routine is called by the kernel during an **I_PUSH ioctl**(2).

Values for *oflag* are the same as those described for the block and character open flags above.

The values for *sflag* are mutually exclusive:

CLONEOPEN	Indicates a clone open (see below.) If the driver supports cloning, it must assign and return a device number of an unused device by changing the value of the device number to which *devp* points.
MODOPEN	Indicates that an **open** routine is being called for a module, not a driver. This is useful in detecting configuration errors and in determining how the driver is being used, since STREAMS drivers can also be configured as STREAMS modules.
0	Indicates a driver is being opened directly, without cloning.

USAGE

This entry point is required in all drivers and STREAMS modules.

The **open** routine could perform any of the following general functions, depending on the type of device and the service provided:

enable device interrupts

allocate buffers or other resources needed to use the device

lock an unsharable device

notify the device of the open

change the device number if this is a clone open

enable put and service procedures for multithreaded drivers

The **open** routine should verify that the minor number component of *devp* is valid, that the type of access requested by *otyp* and *oflag* is appropriate for the device, and, if required, check permissions using the user credentials pointed to by *crp* [see **drv_priv**(D3)].

For STREAMS drivers and modules, the **open** routine is called with interrupts blocked from all STREAMS devices. If the driver sets stream head options by sending an **M_SETOPTS** message upstream from the **open** routine, then the changes are guaranteed to take effect when the system call completes.

Support of cloning is optional. Cloning is the process of the driver selecting an unused device for the user. It eliminates the need to poll many devices when looking for an unused one. Both STREAMS and Non-STREAMS drivers may implement cloning behavior by changing the device number pointed to by *devp*. A driver may designate certain minor devices as special clone entry points into the driver. When these are opened, the driver searches for an unused device and returns the new device number by changing the value of the device number to which *devp* points. Both the major device number and the minor device number can be changed, although usually just the minor number is changed. The major number is only changed when the clone controls more than one device.

Using this method of cloning, a STREAMS driver will never see *sflag* set to **CLONEOPEN**. A different method makes use of this flag. STREAMS drivers can take advantage of a special driver, known as the *clone driver*, to perform clone opens. This frees the driver from having to reserve special minors for the clone entry points. Here, the device node is actually that of the clone driver (the major number is the major number from the clone driver and the minor number is the major number from the real driver.) When the clone driver is opened, it will call the real driver open routine with *sflag* set to **CLONEOPEN**.

For STREAMS drivers and modules, for a given device number (queue), only one instance of the **open** routine can be running at any given time. However, multiple opens on any two different device numbers (queues) can be running concurrently. It is the responsibility of the driver or module to synchronize access to its private data structures in this case. For clone opens, multiple clone opens can run concurrently, and it is the driver's responsibility to synchronize access to its private data structures, as well as allocation and deallocation of device numbers.

Synchronization Constraints

The **open** routine has user context and can sleep. However, STREAMS drivers and modules must sleep such that signals do not cause the sleep to longjump [see **sleep**(D3)].

REFERENCES

close(D2), **drv_priv**(D3), **errnos**(D5), **queue**(D4)

NAME

poll – poll entry point for a non-STREAMS character driver

SYNOPSIS

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

int prefixpoll(dev_t dev, short events, int anyyet, short *reventsp,
      struct pollhead **phpp);
```

Arguments

dev	The device number for the device to be polled.
events	Mask (bit-wise **OR**) indicating the events being polled.
anyyet	A flag that indicates whether the driver should return a pointer to its **pollhead** structure to the caller.
reventsp	A pointer to a bitmask of the returned events satisfied.
phpp	A pointer to a pointer to a **pollhead** structure (defined in **sys/poll.h**).

DESCRIPTION

The **poll** entry point indicates whether certain I/O events have occurred on a given device. It must be provided by any non-STREAMS character device driver that wishes to support polling [see **poll**(2)].

Return Values

The **poll** routine should return 0 for success, or the appropriate error number.

USAGE

This entry point is optional, and is valid for character device drivers only.

Valid values for *events* are:

POLLIN	Data is available to be read (either normal or out-of-band).
POLLOUT	Data may be written without blocking.
POLLPRI	High priority data are available to be read.
POLLHUP	A device hangup.
POLLERR	A device error.
POLLRDNORM	Normal data is available to be read.
POLLWRNORM	Normal data may be written without blocking (same as **POLLOUT**).
POLLRDBAND	Out-of-band data is available to be read.
POLLWRBAND	Out-of-band data may be written without blocking.

A driver that supports polling must provide a **pollhead** structure for each minor device supported by the driver. On systems where they are available, the driver should use the **phalloc**(D3) function to allocate the **pollhead** structure, and use the **phfree**(D3) function to free the **pollhead** structure, if necessary.

The **pollhead** structure must be initialized to zeros prior to its first use (when **phalloc** is used to allocate the structure, this is done automatically).

The definition of the **pollhead** structure is not included in the DDI/DKI, and can change across releases. It should be treated as a ''black box'' by the driver; none of its fields may be referenced. Although the size of the **pollhead** structure is guaranteed to remain the same across releases, it is good practice for drivers not to depend on the size of the structure.

The driver must implement the polling discipline itself. Each time the driver detects a pollable event, it should call **pollwakeup**(D3), passing to it the event that occurred and the address of the **pollhead** structure associated with the device. Note that **pollwakeup** should be called with only one event at a time.

When the driver's **poll** entry point is called, the driver should check if any of the events requested in *events* have occurred. The driver should store the mask, consisting of the subset of **events** that are pending, in the **short** pointed to by *reventsp*. Note that this mask may be 0 if none of the events are pending. In this case, the driver should check the *anyyet* flag and, if it is zero, store the address of the device's **pollhead** structure in the pointer pointed at by *phpp*. The canonical **poll** algorithm is:

```
if (events_are_satisfied_now) {
      *reventsp = events & mask_of_satisfied_events;
} else {
      *reventsp = 0;
      if (!anyyet)
            *phpp = my_local_pollhead_pointer;
}
return (0);
```

Synchronization Constraints

On uniprocessor systems, user context is available in the **poll** routine, but if the driver sleeps, it must do so such that signals do not cause the sleep to longjump [see **sleep**(D3)].

On multiprocessor systems, the **poll** routine may not call any function that sleeps.

REFERENCES

bzero(D3), **phalloc**(D3), **phfree**(D3), **poll**(2), **pollwakeup**(D3), **select**(2)

NAME

print – display a driver message on the system console

SYNOPSIS

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

int prefixprint(dev_t dev, char *str);
```

Arguments

dev	Device number.
str	Pointer to a NULL-terminated character string describing the problem.

DESCRIPTION

The **print** routine is called indirectly by the kernel for the block device when the kernel has detected an exceptional condition (such as out of space) in the device. The driver should print the message on the console along with any driver-specific information.

Return Values

Ignored

USAGE

This entry point is optional, and is valid for block device drivers only.

To display the message on the console, the driver should use the **cmn_err**(D3) function.

The driver should not try to interpret the text string passed to it.

Synchronization Constraints

The **print** routine should not call any functions that sleep.

REFERENCES

cmn_err(D3)

NAME

put – receive messages from the preceding queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
int prefixrput(queue_t *q, mblk_t *mp);   /* read side */
int prefixwput(queue_t *q, mblk_t *mp);   /* write side */
```

Arguments

q	Pointer to the queue.
mp	Pointer to the message block.

DESCRIPTION

The primary task of the **put** routine is to coordinate the passing of messages from one queue to the next in a stream. The **put** routine is called by the preceding component (module, driver, or stream head) in the stream. **put** routines are designated ‘‘write’’ or ‘‘read’’ depending on the direction of message flow.

Return Values

Ignored

USAGE

This entry point is required in all STREAMS drivers and modules.

Both modules and drivers must have write **put** routines. Modules must have read **put** routines, but drivers don’t really need them because their interrupt handler can do the work intended for the read **put** routine. A message is passed to the **put** routine. If immediate processing is desired, the **put** routine can process the message, or it can enqueue it so that the service routine [see **srv**(D2)] can process it later.

The **put** routine must do at least one of the following when it receives a message:

pass the message to the next component in the stream by calling the **putnext**(D3) function

process the message, if immediate processing is required (for example, high priority messages)

enqueue the message with the **putq**(D3) function for deferred processing by the service routine

Typically, the **put** routine will switch on the message type, which is contained in **mp->b_datap->db_type**, taking different actions depending on the message type. For example, a **put** routine might process high priority messages and enqueue normal messages.

The **putq** function can be used as a module’s **put** routine when no special processing is required and all messages are to be enqueued for the service routine.

Although it can be done in the service routine, drivers and modules usually handle queue flushing in their **put** routines.

The canonical flushing algorithm for driver write put routines is as follows:

```
queue_t *q;   /* the write queue */
if (*mp->b_rptr & FLUSHBAND) { /* if driver recognizes bands */
        if (*mp->b_rptr & FLUSHW) {
                flushband(q, FLUSHDATA, *(mp->b_rptr + 1));
                *mp->b_rptr &= ˜FLUSHW;
        }
        if (*mp->b_rptr & FLUSHR) {
                flushband(RD(q), FLUSHDATA, *(mp->b_rptr + 1));
                qreply(q, mp);
        } else {
                freemsg(mp);
        }
} else {
        if (*mp->b_rptr & FLUSHW) {
                flushq(q, FLUSHDATA);
                *mp->b_rptr &= ˜FLUSHW;
        }
        if (*mp->b_rptr & FLUSHR) {
                flushq(RD(q), FLUSHDATA);
                qreply(q, mp);
        } else {
                freemsg(mp);
        }
}
```

The canonical flushing algorithm for module write put routines is as follows:

```
queue_t *q;   /* the write queue */
if (*mp->b_rptr & FLUSHBAND) { /* if module recognizes bands */
        if (*mp->b_rptr & FLUSHW)
                flushband(q, FLUSHDATA, *(mp->b_rptr + 1));
        if (*mp->b_rptr & FLUSHR)
                flushband(RD(q), FLUSHDATA, *(mp->b_rptr + 1));
} else {
        if (*mp->b_rptr & FLUSHW)
                flushq(q, FLUSHDATA);
        if (*mp->b_rptr & FLUSHR)
                flushq(RD(q), FLUSHDATA);
}
if (!SAMESTR(q)) {
        switch (*mp->b_rptr & FLUSHRW) {
        case FLUSHR:
                *mp->b_rptr = (*mp->b_rptr & ˜FLUSHR) | FLUSHW;
                break;
        case FLUSHW:
                *mp->b_rptr = (*mp->b_rptr & ˜FLUSHW) | FLUSHR;
                break;
        }
}
putnext(q, mp);
```

The algorithms for the read side are similar. In both examples, the **FLUSHBAND** flag need only be checked if the driver or module cares about priority bands.

Drivers and modules should not call **put** routines directly.

Drivers should free any messages they do not recognize.

Modules should pass on any messages they do not recognize.

Drivers should fail any unrecognized **M_IOCTL** messages by converting them into **M_IOCNAK** messages and sending them upstream.

Modules should pass on any unrecognized **M_IOCTL** messages.

Synchronization Constraints

put routines do not have user context and so may not call any function that sleeps.

REFERENCES

datab(D4), **flushband**(D3), **flushq**(D3), **msgb**(D4), **putctl**(D3), **putctl1**(D3), **putnext**(D3), **putq**(D3), **qreply**(D3), **queue**(D4), **srv**(D2)

NAME

read – read data from a device

SYNOPSIS

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/cred.h>
#include <sys/ddi.h>
```

int *prefix***read(dev_t** *dev***, uio_t ****uiop***, cred_t ****crp***);**

Arguments

dev	Device number.
uiop	Pointer to the **uio**(D4) structure that describes where the data is to be stored in user space.
crp	Pointer to the user credential structure for the I/O transaction.

DESCRIPTION

The driver **read** routine is called during the **read**(2) system call. The **read** routine is responsible for transferring data from the device to the user data area.

Return Values

The **read** routine should return 0 for success, or the appropriate error number.

USAGE

This entry point is optional, and is valid for character device drivers only.

The pointer to the user credentials, *crp*, is available so the driver can check to see if the user can read privileged information, if the driver provides access to any. The **uio** structure provides the information necessary to determine how much data should be transferred. The **uiomove**(D3) function provides a convenient way to copy data using the **uio** structure.

Block drivers that provide a character interface can use **physiock**(D3) to perform the data transfer with the driver's **strategy**(D2) routine.

Synchronization Constraints

The **read** routine has user context and can sleep.

REFERENCES

drv_priv(D3), **errnos**(D5), **physiock**(D3), **strategy**(D2), **uio**(D4), **uiomove**(D3), **ureadc**(D3), **write**(D2)

NAME

`size` – return size of logical block device

SYNOPSIS

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

int prefixsize(dev_t dev);
```

Arguments

dev	The logical device number.

DESCRIPTION

The **`size`** entry point returns the number of **`NBPSCTR`**-byte units on a logical block device (partition). **`NBPSCTR`**, defined in **`param.h`**, is the number of bytes per logical disk sector.

Return Values

On success, the **`size`** routine should return the number of **`NBPSCTR`**-byte units on the logical block device specified by *dev*; on failure, **`size`** should return –1.

USAGE

This entry point is required in all block device drivers.

`size`(D2) is called only when the device is open.

Synchronization Constraints

The **`size`** routine has user context and can sleep. However, it should be careful not to spend much time sleeping, especially if the routine is called when the logical device is not open.

NAME

srv – service queued messages

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
int prefixrsrv(queue_t *q);   /* read side */
int prefixwsrv(queue_t *q);   /* write side */
```

Arguments

q	Pointer to the queue.

DESCRIPTION

The **srv** (service) routine may be included in a STREAMS module or driver for a number of reasons. It provides greater control over the flow of messages in a stream by allowing the module or driver to reorder messages, defer the processing of some messages, or fragment and reassemble messages. The service routine also provides a way to recover from resource allocation failures.

Return Values

Ignored

USAGE

This entry point is optional, and is valid for STREAMS drivers and modules only.

A message is first passed to a module's or driver's **put**(D2) routine, which may or may not process it. The **put** routine can place the message on the queue for processing by the service routine.

Once a message has been enqueued, the STREAMS scheduler calls the service routine at some later time. Drivers and modules should not depend on the order in which service procedures are run. This is an implementation-dependent characteristic. In particular, applications should not rely on service procedures running before returning to user-level processing.

Every STREAMS queue [see **queue**(D4)] has limit values it uses to implement flow control. High and low water marks are checked to stop and restart the flow of message processing. Flow control limits apply only between two adjacent queues with service routines. Flow control occurs by service routines following certain rules before passing messages along. By convention, high priority messages are not affected by flow control.

STREAMS messages can be defined to have up to 256 different priorities to support some networking protocol requirements for multiple bands of data flow. At a minimum, a stream must distinguish between normal (priority band zero) messages and high priority messages (such as **M_IOCACK**). High priority messages are always placed at the head of the queue, after any other high priority messages already enqueued. Next are messages from all included priority bands, which are enqueued in decreasing order of priority. Each priority band has its own flow control limits. By convention, if a band is flow-controlled, all lower priority bands are also stopped.

Once a service routine is called by the STREAMS scheduler it must provide for processing all messages on its queue, restarting itself if necessary. Message processing must continue until either the queue is empty, the stream is flow-controlled, or an allocation error occurs. Typically, the service routine will switch on the message type, which is contained in **mp->b_datap->db_type**, taking different actions depending on the message type.

For singlethreaded modules and drivers, the framework for the canonical service procedure algorithm is as follows:

```
queue_t *q;
mblk_t *mp;
while ((mp = getq(q)) != NULL) {
      if (mp->b_datap->db_type > QPCTL ||
         canput(q->q_next)) {
            /* process the message */
            putnext(q, mp);
      } else {
            putbq(q, mp);
            return;
      }
}
```

If the singlethreaded module or driver cares about priority bands, the algorithm becomes:

```
queue_t *q;
mblk_t *mp;
while ((mp = getq(q)) != NULL) {
      if (mp->b_datap->db_type > QPCTL ||
         bcanput(q->q_next, mp->b_band)) {
            /* process the message */
            putnext(q, mp);
      } else {
            putbq(q, mp);
            return;
      }
}
```

Each STREAMS module and driver can have a read and write service routine. If a service routine is not needed (because the **put** routine processes all messages), a **NULL** pointer should be placed in the module's **qinit**(D4) structure.

If the service routine finishes running because of any reason other than flow control or an empty queue, then it must explicitly arrange for its rescheduling. For example, if an allocation error occurs during the processing of a message, the service routine can put the message back on the queue with **putbq**, and, before returning, arrange to have itself rescheduled [see **qenable**(D3)] at some later time [see **bufcall**(D3) and **itimeout**(D3)].

Service routines can be interrupted by **put** routines, unless the processor interrupt level is raised.

Only one copy of a queue's service routine will run at a time.

Drivers and modules should not call service routines directly. **qenable**(D3) should be used to schedule service routines to run.

Drivers (excepting multiplexors) should free any messages they do not recognize.

Modules should pass on any messages they do not recognize.

Drivers should fail any unrecognized **M_IOCTL** messages by converting them into **M_IOCNAK** messages and sending them upstream.

Modules should pass on any unrecognized **M_IOCTL** messages.

Service routines should never put high priority messages back on their queues.

Synchronization Constraints

Service routines do not have user context and so may not call any function that sleeps.

REFERENCES

bcanput(D3), **bufcall**(D3), **canput**(D3), **datab**(D4), **getq**(D3), **msgb**(D4), **pcmsg**(D3), **put**(D2), **putbq**(D3), **putnext**(D3), **putq**(D3), **qenable**(D3), **qinit**(D4), **queue**(D4), **itimeout**(D3)

NAME

start – initialize a device at system start-up

SYNOPSIS

void *prefix***start(void);**

DESCRIPTION

The **start** routine is called at system boot time (after system services are available and interrupts have been enabled) to initialize drivers and the devices they control.

Return Values

None

USAGE

This entry point is optional.

The **start** routine can perform the following types of activities:

initialize data structures

allocate buffers for private buffering schemes

map the device into virtual address space

initialize hardware

initialize timeouts

A driver that needs to perform setup and initialization tasks that must take place before system services are available and interrupts are enabled should use the **init**(D2) routine to perform such tasks. The **start** routine should be used for all other initialization tasks.

Synchronization Constraints

Functions that can result in the caller sleeping, or that require user context, such as **sleep**(D3), may not be called from the **start** routine.

REFERENCES

init(D2)

NAME

strategy – perform block I/O

SYNOPSIS

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

int prefixstrategy(struct buf *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **strategy** routine is called by the kernel to read and write blocks of data on the block device. **strategy** may also be called directly or indirectly (via a call to the **physiock**(D3) function) to support the raw character interface of a block device from **read**(D2), **write**(D2) or **ioctl**(D2). The **strategy** routine's responsibility is to set up and initiate the data transfer.

Return Values

Ignored. Errors are returned by using the **bioerror**(D3) function to mark the buffer as being in error. On systems where the **bioerror** function is not available, errors can be returned by setting the **B_ERROR** flag in the **b_flags** field of the **buf** structure, and setting the error number in the **b_error** field of the **buf** structure.

USAGE

This entry point is required in all block device drivers.

Generally, the first validation test performed by the **strategy** routine is to see if the I/O is within the bounds of the device. If the starting block number, given by **bp->b_blkno**, is less than 0 or greater than the number of blocks on the device, the error number in the buffer header should be set to **ENXIO**, and the **B_ERROR** flag should be set in **bp->b_flags**. If the **bioerror** routine is available, **bioerror** should be used to set the buffer error number to **ENXIO**. Then, the buffer should be marked done by calling **biodone**(D3), and the driver should return. If **bp->b_blkno** is equal to the number of blocks on the device and the operation is a write, indicated by the absence of the **B_READ** flag in **bp->b_flags** (**!(bp->b_flags & B_READ)**), then the same action should be taken. However, if the operation is a read and **bp->b_blkno** is equal to the number of blocks on the device, then the driver should set **bp->b_resid** equal to **bp->b_bcount**, mark the buffer done by calling **biodone**, and return. This will cause the read to return 0.

Once the I/O request has been validated, the **strategy** routine should queue the request. If there is not already a transfer under way, the I/O is started. Then the **strategy** routine returns. When the I/O is complete, the driver will call **biodone** to free the buffer and notify anyone who has called **biowait**(D3) to wait for the I/O to finish.

There are two kinds of I/O requests passed to **strategy** routines: normal block I/O requests and paged-I/O requests. Normal block I/O requests are identified by the absence of the **B_PAGEIO** flag or the presence of the **B_MAPPED** in **bp->b_flags**. Here, the starting kernel virtual address of the data transfer will be found in **bp->b_un.b_addr**. Paged-I/O requests are identified by the presence of the **B_PAGEIO** flag in **bp->b_flags**. The driver has several ways to perform a paged-I/O request.

If the driver wants to use virtual addresses, it can call **bp_mapin**(D3) to get a virtually contiguous mapping for the pages. The physical address can be obtained by calling **kvtophys**(D3X) for each page in the virtual address range. However, a more efficient way is to use **bptophys**(D3X) for each page in the list. **bptophys** will return the physical page that corresponds to **bp->b_bcount** minus **bp->b_resid**.

If the amount of data to be transferred is more than can be transferred, the driver can transfer as much as possible (if it supports partial reads and writes), and then use the **bioerror** function to set the buffer error number to EIO. If the **bioerror** function is not available, the driver should then set the **B_ERROR** flag, and set **bp->b_resid** equal to the number of bytes not transferred (if all of the data were transferred, **bp->b_resid** should be set to 0).

Synchronization Constraints

The **strategy** entry point has the necessary context to sleep, but it cannot assume it is called from the same context of the process that initiated the I/O request. Furthermore, the process that initiated the I/O might not even be in existence when the **strategy** routine is called.

REFERENCES

biodone(D3), **bioerror**(D3), **biowait**(D3), **bptophys**(D3X), **bp_mapin**(D3), **buf**(D4), **devflag**(D1), **errnos**(D5), **kvtophys**(D3X), **physiock**(D3), **read**(D2), **write**(D2)

unload(D2)

NAME

`unload` – unload a loadable kernel module

SYNOPSIS

`int` *prefix***`unload(void);`**

DESCRIPTION

The module **`unload`** routine handles any cleanup a loadable kernel module must perform before it can be dynamically unloaded from a running system.

Return Values

The **`unload`** routine should return 0 for success, or the appropriate error number.

USAGE

This entry point is optional.

The **`unload`** routine can perform activities such as:

deallocate memory acquired for private data

cancel any outstanding **`itimeout`**(D3) or **`bufcall`**(D3) requests made by the module

Synchronization Constraints

The **`unload`** routine should not sleep, and should not call any functions that sleep.

NAME

unmap – support virtual unmapping for memory-mapped device

SYNOPSIS

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

int *prefix***unmap(dev_t** *dev***, vhandl_t ***vt***);**

Arguments

dev	Device whose memory is to be mapped.
vt	Handle to caller's virtual address space

DESCRIPTION

To unmap a device, the user program calls **munmap**(2) system call. After performing device-independent unmapping in the user's space, the **munmap** system call calls your driver's *pre***fixunmap** to remove the mapping.

Return Values

On success, 0 is returned. An error number is returned on failure.

USAGE

If a driver provides a **map**(D2X) routine but does not provide an unmapping routine, the munmap system call returns the *ENODEV* error condition to the user. Therefore, it is a good idea for your driver to provide a dummy unmapping routine even if your driver does not need to perform any action to unmap the device.

Synchronization Constraints

The **unmap** routine has user context and can sleep.

REFERENCES

map(D2X), **v_getaddr**(D3X), **v_getlen**(D3X), **v_gethandle**(D3X), **v_mapphys**(D3X)

NAME

write – write data to a device

SYNOPSIS

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/cred.h>
#include <sys/ddi.h>
```

int *prefix***write(dev_t** *dev***, uio_t ***uiop***, cred_t ***crp***);**

Arguments

dev	Device number.
uiop	Pointer to the **uio**(D4) structure that describes where the data is to be fetched from user space.
crp	Pointer to the user credential structure for the I/O transaction.

DESCRIPTION

The driver **write** routine is called during the **write**(2) system call. The **write** routine is responsible for transferring data from the user data area to the device.

Return Values

The **write** routine should return 0 for success, or the appropriate error number.

USAGE

This entry point is optional, and is valid for character device drivers only.

The pointer to the user credentials, *crp*, is available so the driver can check to see if the user can write privileged information, if the driver provides access to any. The **uio** structure provides the information necessary to determine how much data should be transferred. The **uiomove**(D3) function provides a convenient way to copy data using the **uio** structure.

Block drivers that provide a character interface can use **physiock**(D3) to perform the data transfer with the driver's **strategy**(D2) routine.

Synchronization Constraints

The **write** routine has user context and can sleep.

The write operation is intended to be synchronous from the caller's perspective. Minimally, the driver **write** routine should not return until the caller's buffer is no longer needed. For drivers that care about returning errors, the data should be committed to the device. For others, the data might only be copied to local staging buffers. Then the data will be committed to the device asynchronously to the user's request, losing the ability to return an error with the associated request.

REFERENCES

drv_priv(D3), **errnos**(D5), **physiock**(D3), **read**(D2), **strategy**(D2), **uio**(D4), **uiomove**(D3), **uwritec**(D3)

Kernel Utilities and Utility Extensions (D3 and D3X)

NAME

intro – introduction to kernel utility routines

SYNOPSIS

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

DESCRIPTION

This section describes the kernel utility functions available for use by device drivers and STREAMS modules.

USAGE

Drivers and STREAMS modules must not call any kernel routines other than the ones described in this section.

Unless otherwise stated, any kernel utility routine that sleeps will do so such that signals will not interrupt the sleep.

NAME

adjmsg – trim bytes from a message

SYNOPSIS

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

DESCRIPTION

adjmsg removes bytes from a message.

Return Values

If the message can be trimmed successfully, 1 is returned. Otherwise, 0 is returned.

USAGE

|*len*| (the absolute value of *len*) specifies how many bytes are to be removed. If *len* is greater than **0**, bytes are removed from the head of the message. If *len* is less than **0**, bytes are removed from the tail. **adjmsg** fails if |*len*| is greater than the number of bytes in *mp*. If *len* spans more than one message block in the message, the messages blocks must be the same type, or else **adjmsg** will fail.

If *len* is greater than the amount of data in a single message block, that message block is not freed. Rather, it is left linked in the message, and its read and write pointers are set equal to each other, indicating no data present in the block.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

msgb(D4)

NAME

allocb – allocate a message block

SYNOPSIS

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

mblk_t *allocb(int size, uint_t pri);
```

Arguments

size	The number of bytes in the message block.
pri	Priority of the request.

DESCRIPTION

allocb tries to allocate a STREAMS message block.

Return Values

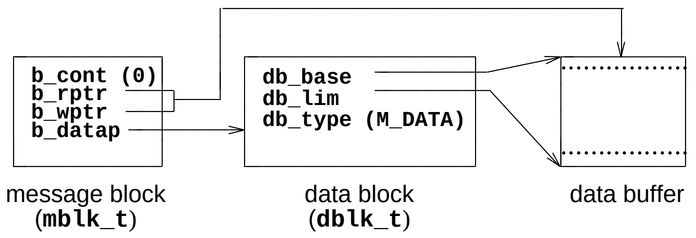
If successful, **allocb** returns a pointer to the allocated message block of type **M_DATA** (defined in **sys/stream.h**). If a block cannot be allocated, a **NULL** pointer is returned.

USAGE

Buffer allocation fails only when the system is out of memory. If no buffer is available, the **bufcall**(D3) function can help a module recover from an allocation failure.

The psi argument is no longer used, but is retained for compatibility.

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



Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

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, 0 is returned, indicating an allocation failure (line 8). Otherwise, the message type is set to **`M_ERROR`** (line 9). Line 10 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 a write queue, the **`q->q_flag`** member is tested to see if **`QREADR`** is set (line 12). If it is not set, *q* points to a write queue, and on line 13 the **`RD`**(D3) function is used to find the corresponding read queue. In line 14, the **`putnext`**(D3) function is used to send the message upstream. Then **`send_error`** returns 1 indicating success.

```
 1  send_error(q, err)
 2    queue_t *q;
 3    uchar_t err;
 4  {
 5    mblk_t *bp;
 6    long fl=0;

 7    if ((bp = allocb(1, BPRI_HI)) == NULL)
 8          return(0);
 9    bp->b_datap->db_type = M_ERROR;
10    *bp->b_wptr++ = err;
11    (void) strqget(q, QFLAG, 0, &fl);
12    if (fl & QREADR))
13          q = RD(q);
14    putnext(q, bp);
15    return(1);
16  }
```

REFERENCES

`bufcall`(D3), **`esballoc`**(D3), **`esbbcall`**(D3), **`freeb`**(D3), **`msgb`**(D4)

NAME

ASSERT – verify assertion

SYNOPSIS

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

void ASSERT(int expression);
```

Arguments

expression	Expression to be evaluated.

DESCRIPTION

ASSERT is a debugging interface for verifying program invariants within code that is compiled with the **DEBUG** compilation option defined.

Return Values

If *expression* evaluates to non-zero, **ASSERT** returns no value. If *expression* evaluates to zero, **ASSERT** panics the system.

USAGE

expression is a boolean expression that the caller expects to evaluate to non-zero (that is, the caller is asserting that the expression has a non-zero value). If *expression* evaluates to non-zero, the **ASSERT** call has no effect. If *expression* evaluates to zero, **ASSERT** causes the system to panic with the following message:

PANIC: assertion failed: *expression*, **file:** *filename*, **line:** *lineno*

where *filename* is the name of the source file in which the failed assertion appears and *lineno* is the line number of the **ASSERT** call within the file.

When the **DEBUG** compilation option is not defined, **ASSERT** calls are not compiled into the code, and therefore have no effect, including the fact that *expression* is not evaluated.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

cmn_err(D3)

NAME

badaddr – check for bus error when reading an address

SYNOPSIS

badaddr(char ****addr*, int** *size***);**

Arguments

addr	The address of the location to be read.
size	The size in bytes of the location to be read. *size* can be: 1 (one byte), 2 (two bytes equals short or half word), or 4 (four bytes equals long word).

DESCRIPTION

Call **badaddr** to determine whether you can read the specified address location. Typically, you call **badaddr** from a VME device's **edtinit()** function to determine whether a device is still on the present system.

Return Values

If the addressed location is accessible, **badaddr** returns 0. Otherwise, **badaddr** returns 1.

See Also

wbadaddr(D3X)

NAME

bcanput – test for flow control in a specified priority band

SYNOPSIS

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

int bcanput(queue_t *q, uchar_t pri);
```

Arguments

q	Pointer to the message queue.
pri	Message priority.

DESCRIPTION

Like the **canput**(D3) function, **bcanput** searches through the stream (starting at *q*) until it finds a queue containing a service routine, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if a message of priority *pri* can be enqueued. If the band is full, **bcanput** marks the queue to automatically back-enable the caller's service routine when the amount of data in messages on the queue has reached its low water mark.

Return Values

bcanput returns 1 if a message of priority *pri* can be sent in the stream, or 0 if the priority band is flow-controlled. If **bcanput** reaches the end of the stream without finding a queue with a service routine, then it returns 1.

USAGE

The driver is responsible for both testing a queue with **bcanput** and refraining from placing a message on the queue if **bcanput** fails.

It is possible because of race conditions to test for room using **bcanput** and get an indication that there is room for a message, and then have the queue fill up before subsequently enqueuing the message, causing a violation of flow control. This is not a problem, since the violation of flow control in this case is bounded.

If *pri* is 0, the **bcanput** call is equivalent to a call to **canput**.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

canput(D3), **putbq**(D3)

NAME

bcopy – copy data between address locations in the kernel

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ddi.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 the copy is made.
bcount	Number of bytes to be copied.

DESCRIPTION

bcopy copies *bcount* bytes from one kernel address to another. It chooses the best algorithm based on address alignment and number of bytes to copy.

Return Values

None

USAGE

If the input and output addresses overlap, the function executes, but the results are undefined.

The source and destination address ranges must both be within the kernel address space and must be memory resident. No range checking is done. Since there is no mechanism by which drivers that conform to the rules of the DDI/DKI can obtain and use a kernel address which is not memory resident (an address which is paged out), DDI/DKI conforming drivers can assume that any address to which they have access is memory resident and therefore a valid argument to **bcopy**. Addresses within user address space are not valid arguments, and specifying such an address may cause the driver to corrupt the system in an unpredictable way. For copying between kernel and user space, drivers must use an appropriate function defined for that purpose (for example, **copyin**(D3), **copyout**(D3), **uiomove**(D3), **ureadc**(D3), or **uwritec**(D3)).

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

Examples

An I/O request is made for data stored in a RAM disk. If the I/O operation is a read request, data are copied from the RAM disk to a buffer (line 9). If it is a write request, data are copied from a buffer to the RAM disk (line 15). The **bcopy** function is used since both the RAM disk and the buffer are part of the kernel address space.

```
#define RAMDNBLK    1000           /* number of blocks in RAM disk */
#define RAMDBSIZ    NBPSCTR        /* bytes per block */
char ramdblks[RAMDNBLK][RAMDBSIZ]; /* blocks forming RAM disk */
   ...

if (bp->b_flags & B_READ) {
        /*
         * read request - copy data from RAM disk to system buffer
         */
        bcopy(ramdblks[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);

} else {
        /*
         * write request - copy data from system buffer to RAM disk
         */
        bcopy(bp->b_un.b_addr, ramdblks[bp->b_blkno], bp->b_bcount);
}
```

REFERENCES

copyin(D3), **copyout**(D3), **uiomove**(D3), **ureadc**(D3), **uwritec**(D3)

NAME

biodone – release buffer after block I/O and wakeup processes

SYNOPSIS

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

void biodone(buf_t *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **biodone** function is called by the driver to indicate that block I/O associated with the buffer header *fp* is complete, and that it can be reused.

Return Values

None

USAGE

biodone is usually called from the driver's **strategy**(D2) routine or I/O completion handler [usually **intr**(D2)].

If the driver (or the kernel) had specified an iodone handler by initializing the **b_iodone** field of the **buf**(D4) structure to the address of a function, that function is called with the single argument, *bp*. Then **biodone** returns.

If an iodone handler had not been specified, **biodone** sets the **B_DONE** flag in the **b_flags** field of the buffer header. Then, if the **B_ASYNC** flag is set, the buffer is released back to the system. If the **B_ASYNC** flag is not set, any processes waiting for the I/O to complete are awakened.

If the buffer was allocated via **getrbuf**(D3), the driver must have specified an iodone handler.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

Examples

Generally, the first validation test performed by any block device **strategy** routine is a check to verify the bounds of the I/O request. If a **read** request is made for one block beyond the limits of the device (line 8), it will report an end-of-media condition (line 10). Otherwise, if the request is outside the limits of the device, the routine will report an error condition (line 12). In either case, the I/O operation is completed by calling **biodone** (line 14) and the driver returns.

```
 1  #define RAMDNBLK1000          /* 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  ramdstrategy(bp)
 5     struct buf *bp;
 6  {
 7     daddr_t blkno = bp->b_blkno;

 8     if ((blkno < 0) || (blkno >= RAMDNBLK)) {
 9            if ((blkno == RAMDNBLK) && (bp->b_flags & B_READ)) {
10                   bp->b_resid = bp->b_bcount;      /* nothing read */
11            } else {
12                   bioerror(bp, ENXIO);
13            }
14            biodone(bp);
15            return;
16     }
        . . .
```

On systems where the function **bioerror**(D3) is not available, line 12 could read:

```
bp->b_error = ENXIO; bp->b_flags |= B_ERROR;
```

REFERENCES

bioerror(D3), **biowait**(D3), **brelse**(D3), **buf**(D4), **freerbuf**(D3), **getrbuf**(D3), **intr**(D2), **strategy**(D2)

bioerror(D3)

NAME

bioerror – manipulate error fields within a buffer header

SYNOPSIS

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

void bioerror(buf_t *bp, int errno);
```

Arguments

bp	Pointer to the buffer header structure.
errno	Error number to be set, or zero to indicate that the error fields within the buffer header should be cleared.

DESCRIPTION

bioerror is used to manipulate the error fields within a buffer header (**buf**(D4) structure).

Return Values

None

USAGE

Driver code (for example, a **strategy**(D2) routine) that wishes to report an I/O error condition associated with the buffer pointed to by *bp* should call **bioerror** with *errno* set to the appropriate error number. This will set the appropriate fields within the buffer header so that higher level code can detect the error and retrieve the error number using **geterror**(D3).

The error fields within the buffer header can be cleared by calling **bioerror** with *errno* set to zero.

On multiprocessor systems, DDI/DKI conforming drivers are no longer permitted to manipulate the error fields of the **buf** structure directly. **bioerror** must be used for this purpose.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

REFERENCES

buf(D4), **errnos**(D5), **geteblk**(D3), **geterror**(D3), **getrbuf**(D3), **ngeteblk**(D3), **strategy**(D2)

NAME

`biowait` – suspend processes pending completion of block I/O

SYNOPSIS

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

int biowait(buf_t *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **`biowait`** function suspends process execution during block I/O.

Return Values

If an error occurred during the I/O transfer, the error number is returned. Otherwise, on success, 0 is returned.

USAGE

Block drivers that have allocated their own buffers with **`geteblk`**(D3), **`getrbuf`**(D3), or **`ngeteblk`**(D3) can use **`biowait`** to suspend the current process execution while waiting for a read or write request to complete.

Drivers using **`biowait`** must use **biodone**(D3) in their I/O completion handlers to signal **`biowait`** when the I/O transfer is complete.

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read/write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

REFERENCES

biodone(D3), **`buf`**(D4), **`geteblk`**(D3), **`getrbuf`**(D3), **`intr`**(D2), **`ngeteblk`**(D3), **`strategy`**(D2)

NAME

bptophys – get physical address of buffer data

SYNOPSIS

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

paddr_t bptophys(void *bp);
```

Arguments

bp	Pointer to buffer header structure.

DESCRIPTION

This function returns a pointer to the physical address of the data mapped starting at *bp->b_bcount* minus *bp->b_resid*. The driver routine must set *b_resid* to the number of bytes outstanding to transfer before calling **bptophys**. The returned value is valid only up to the next page boundary.

Return Values

On success, a pointer to the physical address of the mapped page is returned. If the end of the list is reached, *NULL* is returned.

Level

Base or Interrupt.

Notes

Does not sleep.

Driver defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

See Also

getpagesize(2), **strategy**(D2), **bp_mapin**(D3), **bp_mapout**(D3), **buf**(D4)

Warnings

This interface is deprecated; the function **getnextpg**(D3X) should be used to access the data of a paged-I / O buffer header.

NAME

`bp_mapin` – allocate virtual address space for buffer page list

SYNOPSIS

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

void bp_mapin(struct buf_t *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **`bp_mapin`** function is used to map virtual address space to a page list maintained by the buffer header [see **`buf`**(D4)] during a paged-I/O request.

Return Values

None

USAGE

A paged-I/O request is identified by the **`B_PAGEIO`** flag being set in the **`b_flags`** field of the buffer header passed to a driver's **`strategy`**(D2) routine.

`bp_mapin` allocates system virtual address space, maps that space to the page list, and returns the new virtual address in the **`b_un.b_addr`** field of the **`buf`** structure. This address is the virtual address of the start of the page mappings, plus the offset given by the original value of **`bp->b_un.b_addr`**. After the I/O completes, the virtual address space can be deallocated using the **`bp_mapout`**(D3) function.

Level

Base only.

Synchronization Constraints

This routine may sleep if virtual space is not immediately available.

Driver-defined basic locks and read/write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

REFERENCES

`bp_mapout`(D3), **`buf`**(D4), **`strategy`**(D2)

NAME

bp_mapout – deallocate virtual address space for buffer page list

SYNOPSIS

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

void bp_mapout(struct buf_t *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **bp_mapout** function deallocates the system virtual address space associated with a buffer header page list.

Return Values

None

USAGE

The virtual address space must have been allocated by a previous call to **bp_mapin**(D3). Drivers should not reference any virtual addresses in the mapped range after **bp_mapout** has been called.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

bp_mapin(D3), **buf**(D4)

NAME

`brelse` – return a buffer to the system's free list

SYNOPSIS

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

void brelse(struct buf_t *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **`brelse`** function returns the buffer specified by *bp* to the system's buffer free list. If there were any processes waiting for this specific buffer to become free, or for any buffer to become available on the free list, one is awakened.

Return Values

None

USAGE

The buffer specified by *bp* must have been previously allocated by a call to **`geteblk`**(D3) or **`ngeteblk`**(D3). **`brelse`** may not be called to release a buffer which has been allocated by any other means.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`biodone`(D3), **`biowait`**(D3), **`buf`**(D4), **`clrbuf`**(D3), **`geteblk`**(D3), **`ngeteblk`**(D3)

NAME

btod – convert from bytes to disk sectors

SYNOPSIS

```
#include "sys/types.h"
#include "sys/param.h"
#include "sys/immu.h"

btod(int num_bytes);
```

DESCRIPTION

btod is a macro that converts from a byte count to a disk sector count, where a disk sector is defined as 512 bytes. The **btod** macro rounds the byte count up to a full sector size before conversion.

NAME

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

SYNOPSIS

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

ulong_t btop(ulong_t numbytes);
```

Arguments

numbytes	Size in bytes to convert to equivalent size in pages.

DESCRIPTION

`btop` returns the number of pages that are contained in the specified number of bytes, with downward rounding if the byte count is not a page multiple.

Return Values

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

USAGE

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

If the page size is 2048, then **`btop(4096)`** and **`btop(4097)`** both return 2, and **`btop(4095)`** returns 1. **`btop(0)`** returns 0.

REFERENCES

`btopr`(D3), **`ptob`**(D3)

NAME

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

SYNOPSIS

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

ulong_t btopr(ulong_t numbytes);
```

Arguments

numbytes	Size in bytes to convert to equivalent size in pages.

DESCRIPTION

`btopr` returns the number of pages that are contained in the specified number of bytes, with upward rounding if the byte count is not a page multiple.

Return Values

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

USAGE

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

If the page size is 2048, then **`btopr(4096)`** and **`btopr(4095)`** both return 2, and **`btopr(4097)`** returns 3. **`btopr(0)`** returns 0.

REFERENCES

`btop`(D3), **`ptob`**(D3)

NAME

bufcall – call a function when a buffer becomes available

SYNOPSIS

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

toid_t bufcall(uint_t *size*, **int** *pri*, **void (***func**)(),** **long** *arg***);**

Arguments

size	Number of bytes in the buffer to be allocated (from the failed **allocb**(D3) request).
pri	Priority of the **allocb** allocation request.
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

When a buffer allocation request fails, the function **bufcall** can be used to schedule the routine, *func*, to be called with the argument, *arg*, when a buffer of at least *size* bytes becomes available. **bufcall** serves, in effect, as a timeout call of indeterminate length.

Return Values

On success, **bufcall** returns a non-zero value that identifies the scheduling request. On failure, **bufcall** returns 0.

USAGE

When *func* runs, all interrupts from STREAMS devices will be blocked. On multiprocessor systems, when *func* runs all interrupts from STREAMS devices will be blocked on the processor on which *func* is running. *func* will have no user context and may not call any function that sleeps.

Even when *func* is called, **allocb** can still fail if another module or driver had allocated the memory before *func* was able to call **allocb**.

The *pri* argument is no longer used but is retained for compatibility.

The non-zero identifier returned by **bufcall** may be passed to **unbufcall**(D3) to cancel the request.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Example

The purpose of this service routine [see **srv**(D2)] is to add a header to all **M_DATA** messages. We assume only **M_DATA** messages are added to its queue. Service routines must process all messages on their queues before returning, or arrange to be rescheduled.

While there are messages to be processed (line 19), we check to see if we can send the message on in the stream. If not, we put the message back on the queue (line 21) and return. The STREAMS flow control mechanism will re-enable us later when messages can be sent. If **canput**(D3) succeeded, we try to allocate a buffer large enough to hold the header (line 24). If no buffer is available, the service routine must be rescheduled later, when a buffer is available. We put the original message back on the queue (line 26) and use **bufcall** to attempt the rescheduling (lines 27 and 28). If **bufcall** succeeds, we set the **m_type** field in the module's private data structure to **BUFCALL**. If **bufcall** failed, we use **itimeout**(D3) to reschedule us instead (line 30). **modcall** will be called in about a half second [**drv_usectohz(500000)**]. When the rescheduling has been done, we return.

When **modcall** runs, it will set the **m_type** field to zero, indicating that there is no outstanding request. Then the queue's service routine is scheduled to run by calling **qenable**(D3).

If the buffer allocation is successful, we initialize the header (lines 37–39), make the message type **M_PROTO** (line 41), link the **M_DATA** message to it (line 42), and pass it on (line 43).

See **unbufcall**(D3) for the other half of this example.

```
 1  struct hdr {
 2    uint_t    h_size;
 3    int    h_version;
 4  };
 5  struct mod {
 6    long    m_id;
 7    char    m_type;
    ...
 8  };
 9  #define TIMEOUT    1
10  #define BUFCALL    2
    ...
11  modsrv(q)    /* assume only M_DATA messages enqueued here */
12        queue_t *q;
13  {
14    mblk_t *bp;
15    mblk_t *mp;
16    struct hdr *hp;
17    struct mod *modp;

18    modp = (struct mod *)q->q_ptr;
19    while ((mp = getq(q)) != NULL) {
20        if (!canput(q->q_next)) {
21            putbq(q, mp);
22            return;
23        }
24        bp = allocb(sizeof(struct hdr), BPRI_MED);
25        if (bp == NULL) {
26            putbq(q, mp);
27            modp->m_id = bufcall(sizeof(struct hdr), BPRI_MED,
                               modcall, (long)q);
          if (modp->m_id == 0) {
             modp->m_id = itimeout(modcall, (long)q,
                                   drv_usectohz(500000), plstr);
             modp->m_type = TIMEOUT;
          } else {
             modp->m_type = BUFCALL;
          }
          return;
      }
      hp = (struct hdr *)bp->b_wptr;
      hp->h_size = msgdsize(mp);
      hp->h_version = 1;
      bp->b_wptr += sizeof(struct hdr);
      bp->b_datap->db_type = M_PROTO;
      bp->b_cont = mp;
      putnext(q, bp);
  }
}

modcall(q)
   queue_t *q;
{
   struct mod *modp;

   modp = (struct mod *)q->q_ptr;
   modp->m_type = 0;
   qenable(q);
}
```

REFERENCES

allocb(D3), **esballoc**(D3), **esbbcall**(D3), **itimeout**(D3), **unbufcall**(D3)

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.

DESCRIPTION

The **bzero** function clears a contiguous portion of memory by filling the memory with zeros. It chooses the best algorithm based on address alignment and number of bytes to clear.

Return Values

None

USAGE

There are no alignment restrictions on *addr*, and no length restrictions on *bytes*, other than the address range specified must be within the kernel address space and must be memory resident. No range checking is done. Since there is no mechanism by which drivers that conform to the rules of the DDI / DKI can obtain and use a kernel address that is not memory resident (an address that is paged out), DDI / DKI conforming drivers can assume that any address to which they have access is memory resident and therefore a valid argument to **bzero**. An address within user address space is not a valid argument, and specifying such an address may cause the driver to corrupt the system in an unpredictable way.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

In a driver **close**(D2) routine, rather than clear each individual member of its private data structure, the driver could use **bzero** as shown here:

```
bzero((caddr_t)&drv_dat[getminor(dev)], sizeof(struct drvr_data));
```

REFERENCES

bcopy(D3), **clrbuf**(D3), **kmem_zalloc**(D3)

NAME

canput – test for room in a message queue

SYNOPSIS

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

int canput(queue_t *q);
```

Arguments

q	Pointer to the message queue.

DESCRIPTION

canput tests if there is room for a message in the queue pointed to by *q*. The queue must have a service procedure.

Return Values

canput returns 1 if a message can be placed on the queue. 0 is returned if a message cannot be enqueued because of flow control.

USAGE

The driver is responsible for both testing a queue with **canput** and refraining from placing a message on the queue if **canput** fails.

It is possible because of race conditions to test for room using **canput** and get an indication that there is room for a message, and then have the queue fill up before subsequently enqueuing the message, causing a violation of flow control. This is not a problem, since the violation of flow control in this case is bounded.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

Examples

See **bufcall**(D3) for an example of **canput**.

REFERENCES

bcanput(D3), **putbq**(D3), **putnext**(D3)

NAME

clrbuf – erase the contents of a buffer

SYNOPSIS

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

void clrbuf(buf_t *bp);
```

Arguments

bp	Pointer to the buffer header structure.

DESCRIPTION

The **clrbuf** function zeros a buffer and sets the **b_resid** member of the **buf**(D4) structure to 0. Zeros are placed in the buffer starting at the address specified by **b_un.b_addr** for a length of **b_bcount** bytes.

Return Values

None

USAGE

If the buffer has the **B_PAGEIO** or the **B_PHYS** flag set in the **b_flags** field, then **clrbuf** should not be called until the proper virtual space has been allocated by a call to **bp_mapin**(D3).

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

bp_mapin(D3), **buf**(D4)

NAME

cmn_err – display an error message or panic the system

SYNOPSIS

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

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

Arguments

level	Indicates the severity of the error condition.
format	The message to be displayed.
args	The set of arguments passed with the message being displayed.

DESCRIPTION

cmn_err displays a specified message on the console and / or stores it in the kernel buffer **putbuf**. **cmn_err** can also panic the system.

Return Values

None

USAGE

level Argument

Valid values for *level* are:

CE_CONT Used to continue a previous 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 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:** ,’’ and panic the system. Drivers should use this level only for debugging or in the case of severe errors that indicate that the system cannot continue to function. This level halts processing.

format Argument

By default, the message is sent both to the system console and to the circular kernel buffer **putbuf**. If the first character in *format* is an exclamation point (‘‘**!**’’), the message goes only to **putbuf**. If the first character in *format* is a circumflex (‘‘ˆ’’), the message goes only to the console. The size of the kernel buffer **putbuf** is defined by the kernel variable **putbufsz**. Driver developers or administrators can read the **putbuf** buffer using appropriate debugging or administrative tools [for example, **idbg**(1M)].

cmn_err appends **\n** to each *format* string, even when a message is sent to **putbuf**, except when *level* is **CE_CONT**.

Valid conversion specifications are %**s**, %**u**, %**d**, %**o**, and %**x**. The **cmn_err** function is otherwise similar to the **printf**(3S) library subroutine in its interpretation of the *format* string, except that **cmn_err** does not accept length specifications in conversion specifications. For example, %**3d** is invalid and will be treated as a literal string, resulting in a mismatch of arguments.

args Argument

Any argument within the range of supported conversion specifications can be passed.

General Considerations

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

The **cmn_err** function 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.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

If *level* is **CE_PANIC**, then driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function. For other levels, locks may be held.

Examples

The **cmn_err** function can record tracing and debugging information only in the **putbuf** buffer (lines 12 and 13) or display problems with a device only on the system console (lines 17 and 18).

```
 1  struct  device {                   /* device registers layout */
        ...
 2      int status;                   /* device status word */
 3  };
 4  extern struct device xx_dev[];     /* physical device registers */
 5  extern int xx_cnt;                 /* number of physical devices */
    ...
 6  int
 7  xxopen(dev_t *devp, int flag, int otyp, cred_t *crp)
 8  {
 9      struct device *dp;
10      dp = xx_dev[getminor(*devp)];  /* get dev registers */
11  #ifdef DEBUG                       /* in debugging mode, log function call */
12      cmn_err(CE_NOTE, "!xxopen function call, dev = 0x%x", *devp);
13      cmn_err(CE_CONT, "! flag = 0x%x", flag);
14  #endif
15      /* display device power failure on system console */
16      if ((dp->status & POWER) == OFF)
17              cmn_err(CE_WARN, "^xxopen: Power is OFF on device %d port %d",
18                 getemajor(*devp), geteminor(*devp));
```

REFERENCES

print(D2), **printf**(3S)

copyb(D3)

NAME

copyb – copy a message block

SYNOPSIS

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

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

Arguments

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

DESCRIPTION

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

Return Values

On success, **copyb** returns a pointer to the newly allocated message block containing the copied data. On failure, it returns a **NULL** pointer.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Example

This example illustrates how **copyb** can be used during message retransmission. If there are no messages to retransmit, we return (line 18). For each retransmission record in the list, we test to see if the downstream queue is full with the **canput**(D3) function (line 21). If it is full, we skip the current retransmission record and continue searching the list. If it is not full, we use **copyb**(D3) to copy a header message block (line 25), and **dupmsg**(D3) to duplicate the data to be retransmitted (line 28). If either operation fails, we clean up and break out of the loop.

Otherwise, we update the new header block with the correct destination address (line 34), link the message to be retransmitted to it (line 35), and send it downstream (line 36). At the end of the list, we reschedule a **itimeout** at the next valid interval (line 39) and return.

```
 1  struct retrns {
 2     mblk_t         *r_mp;       /* message to retransmit */
 3     long            r_address;  /* destination address */
 4     queue_t        *r_outq;     /* output queue */
 5     struct retrns  *r_next;     /* next retransmission */
 6  };
 7  struct protoheader {
 8     long            h_address;  /* destination address */
```

```
   ...
};
mblk_t *header;
struct retrns *rlist;
...
retransmit()
{
   mblk_t *bp, *mp;
   struct retrns *rp;
   struct protoheader *php;

   if (!rlist)
          return;
   rp = rlist;
   while (rp) {
          if (!canput(rp->r_outq->q_next)) {
                 rp = rp->r_next;
                 continue;
          }
          bp = copyb(header);
          if (bp == NULL)
                 break;
          mp = dupmsg(rp->r_mp);
          if (mp == NULL) {
                 freeb(bp);
                 break;
          }
          php = (struct protoheader *)bp->b_rptr;
          php->h_address = rp->r_address;
          bp->bp_cont = mp;
          putnext(rp->r_outq, bp);
          rp = rp->r_next;
   }
   (void) itimeout(retransmit, 0, RETRNS_TIME, plstr);
}
```

REFERENCES

allocb(D3), **copymsg**(D3), **msgb**(D4)

copyin(D3)

NAME

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

SYNOPSIS

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

int copyin(caddr_t userbuf, caddr_t driverbuf, size_t count);
```

Arguments

userbuf	User source address from which copy is made.
driverbuf	Driver destination address to which copy is made.
count	Number of bytes to copy.

DESCRIPTION

copyin copies *count* bytes of data from the user virtual address specified by *userbuf* to the kernel virtual address specified by *driverbuf*.

Return Values

If the copy is successful, 0 is returned. Otherwise, –1 is returned to indicate that the specified user address range is not valid.

USAGE

The driver must ensure that adequate space is allocated for the destination address.

copyin chooses the best algorithm based on address alignment and number of bytes to copy. Although the source and destination addresses are not required to be word aligned, word aligned addresses may result in a more efficient copy.

Drivers usually convert a return value of –1 into an **EFAULT** error.

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read/write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

Warnings

The driver source buffer must be completely within the kernel address space, or the system can panic.

When holding sleep locks across calls to this function, multithreaded drivers must be careful to avoid creating a deadlock. During the data transfer, page fault resolution might result in another I/O to the same device. For example, this could occur if the driver controls the disk drive used as the swap device.

Examples

A driver **ioctl**(D2) routine (line 5) can be used to get or set device attributes or registers. If the specified command is **XX_SETREGS** (line 9), the driver copies user data to the device registers (line 11). If the user address is invalid, an error code is returned.

```
 1  struct  device {   /* device registers layout */
        ...
 2      int command;   /* device command word */
 3  };
 4  extern struct device xx_dev[];    /* physical device registers */
    ...
 5  xxioctl(dev_t dev, int cmd, void *arg, int mode, cred_t *crp, int *rvp)
 6  {
 7      struct device *dp;
 8      switch (cmd) {
 9      case XX_SETREGS:       /* copy user program data to device registers */
10              dp = &xx_dev[getminor(dev)];
11              if (copyin(arg, (caddr_t)dp, sizeof(struct device)))
12                      return (EFAULT);
13              break;
```

REFERENCES

bcopy(D3), **copyout**(D3), **uiomove**(D3), **ureadc**(D3), **uwritec**(D3)

copymsg(D3)

NAME

copymsg – copy a message

SYNOPSIS

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

mblk_t *copymsg(mblk_t *mp);
```

Arguments

mp	Pointer to the message to be copied.

DESCRIPTION

copymsg forms a new message by allocating new message blocks, copies the contents of the message referred to by *mp* (using the **copyb**(D3) function), and returns a pointer to the new message.

Return Values

On success, **copymsg** returns a pointer to the new message. On failure, it returns a **NULL** pointer.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

The routine **lctouc** converts all the lower case ASCII characters in the message to upper case. 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 **copymsg** fails (line 9), we return **NULL** (line 10). Otherwise, we free the original message (line 11). If the reference count was equal to one, the message can be modified. For each character (line 16) in each message block (line 15), if it is a lower case letter, we convert it to an upper case letter (line 18). When done, we return a pointer to the converted message (line 21).

```
 1  mblk_t *lctouc(mp)
 2     mblk_t *mp;
 3  {
 4     mblk_t *cmp;
 5     mblk_t *tmp;
 6     uchar_t *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     }
```

```
    for (tmp = cmp; tmp; tmp = tmp->b_next) {
            for (cp = tmp->b_rptr; cp < tmp->b_wptr; cp++) {
                    if ((*cp <= ’z’) && (*cp >= ’a’))
                            *cp -= 0x20;
            }
    }
    return(cmp);
}
```

REFERENCES

allocb(D3), **copyb**(D3), **msgb**(D4)

NAME

`copyout` – copy data from a driver buffer to a user buffer

SYNOPSIS

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

int copyout(caddr_t driverbuf, caddr_t userbuf, size_t count);
```

Arguments

driverbuf	Driver source address from which copy is made.
userbuf	User destination address to which copy is made.
count	Number of bytes to copy.

DESCRIPTION

`copyout` copies *count* bytes of data from the kernel virtual address specified by *driverbuf* to the user virtual address specified by *userbuf*.

Return Values

On success, **`copyout`** returns 0. On failure, it returns –1 to indicate that the specified user address range is not valid.

USAGE

`copyout` chooses the best algorithm based on address alignment and number of bytes to copy. Although the source and destination addresses are not required to be word aligned, word aligned addresses may result in a more efficient copy.

Drivers usually convert a return value of –1 into an **`EFAULT`** error.

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

Warnings

The driver source buffer must be completely within the kernel address space, or the system can panic.

When holding sleep locks across calls to this function, drivers must be careful to avoid creating a deadlock. During the data transfer, page fault resolution might result in another I / O to the same device. For example, this could occur if the driver controls the disk drive used as the swap device.

Examples

A driver **`ioctl`**(D2) routine (line 5) can be used to get or set device attributes or registers. If the specified command is **`XX_GETREGS`** (line 9), the driver copies the current device register values to a user data area (line 11). If the user address is invalid, an error code is returned.

```
struct  device {                 /* device registers layout */
...
   int status;                   /* device status word */
};
extern struct device xx_dev[];   /* physical device registers */
...
xxioctl(dev_t dev, int cmd, void *arg, int mode, cred_t *crp, int *rvp)
{
   struct device *dp;
   switch (cmd) {
   case XX_GETREGS:       /* copy device registers to user program */
          dp = &xx_dev[getminor(dev)];
          if (copyout((caddr_t)dp, arg, sizeof(struct device)))
                 return (EFAULT);
          break;
```

REFERENCES

bcopy(D3), **copyin**(D3), **uiomove**(D3), **ureadc**(D3), **uwritec**(D3)

NAME

cpsema – conditionally perform a "P" or wait semaphore operation

SYNOPSIS

```
#include  "sys/types.h"
#include  "sys/sema.h"

cpsema(sema_t  *semap);
```

Arguments

semap Expects a pointer to the semaphore you want **cpsema** to conditionally decrement.

DESCRIPTION

cpsema conditionally performs a "P" operation depending on the current value of the semaphore. If the semaphore value is less than or equal to 0, **cpsema** returns without altering the semaphore. Otherwise, **cpsema** decrements the semaphore value by 1 and returns. **cpsema** effectively performs a "P" operation if it does not cause the process to sleep; otherwise, it simply returns. To initialize semaphores before using them, call **initnsema**(D3X) or **initnsema_mutex**(D3X).

Return Values

cpsema returns 0 if the semaphore value is less than or equal to 0. (If **cpsema** returns 0, **psema**(D3X) would have slept.) Otherwise **cpsema** returns 1.

See Also

initnsema(D3X), **initnsema_mutex**(D3X), **psema**(D3X), **SLEEP_TRYLOCK**(D3)

NAME

cvsema – conditionally perform a "V" or wait semaphore operation

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"

cvsema(sema_t *semap);
```

Arguments

semap	Expects a pointer to the semaphore you want to conditionally.

DESCRIPTION

cvsema routine conditionally performs a "V" operation depending on the current value of the semaphore. If the semaphore value is strictly less than 0, **cvsema** increments the semaphore value by 1 and wakes up a sleeping process. Otherwise, **cvsema** simply returns. **cvsema** effectively performs a "V" operation if there is a process asleep on the semaphore; otherwise, it does nothing. To initialize semaphores before you use them, call **initnsema**(D3X) or **initnsema_mutex**(D3X).

Return Values

cvsema returns 1 if the semaphore value is less than 0 and a process is awakened. Otherwise **cvsema** returns 0.

See Also

initnsema(D3X), **initnsema_mutex**(D3X), **vsema**(D3X)

NAME

datamsg – test whether a message is a data message

SYNOPSIS

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

int datamsg(uchar_t type);
```

Arguments

type	The type of message to be tested.

DESCRIPTION

The **datamsg** function 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 if the message is any other type.

USAGE

The **db_type** field of the **datab** structure contains the message type. This field may be accessed through the message block using **mp->b_datap->db_type**.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

The **put**(D2) routine enqueues all data messages for handling by the **srv**(D2) (service) routine. All non-data messages are handled in the **put** 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:
   ...
   }
}
```

REFERENCES

allocb(D3), **datab**(D4), **messages**(D5), **msgb**(D4)

delay(D3)

NAME

delay – delay process execution for a specified number of clock ticks

SYNOPSIS

void delay(long *ticks***);**

Arguments

ticks	The number of clock ticks to delay.

DESCRIPTION

delay causes the caller to sleep for the amount of time specified by *ticks*, which is in units of clock ticks. The exact length of the delay is not guaranteed but it will not be less than *ticks -1* clock ticks.

Return Values

None

USAGE

The length of a clock tick can vary across different implementations and therefore drivers should not include any hard-coded assumptions about the length of a tick. The **drv_usectohz**(D3) and **drv_hztousec**(D3) functions can be used, as necessary, to convert between clock ticks and microseconds (implementation independent units).

The **delay** function calls **itimeout**(D3) to schedule a wakeup after the specified amount of time has elapsed. **delay** then goes to sleep until **itimeout** wakes up the sleeping process.

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read/write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function, but this is discouraged because it can adversely affect performance by forcing any other processes contending for the lock to sleep for the duration of the delay.

REFERENCES

drv_hztousec(D3), **drv_usectohz**(D3), **drv_usecwait**(D3), **itimeout**(D3), **sleep**(D3), **untimeout**(D3), **wakeup**(D3)

NAME

dki_dcache_inval – invalidate the data cache for a given range of virtual addresses

SYNOPSIS

```
#include "sys/types.h"

dki_dcache_inval(caddr_t v_addr, unsigned len);
```

Arguments

v_addr	Can be either a user or kernel virtual address. If *v_addr* is a user virtual address, it is assumed to be that of the current mapped process. If, however, *v_addr* is a *k1seg* address, or if it is a user virtual or *k2seg* address and the page table entry specifies that the page is not cacheable, no operation is performed on the data cache for that page.
len	Gives the number of bytes over which to perform the operation.

DESCRIPTION

dki_dcache_inval invalidates the data cache starting at *v_addr* address. This function, along with the **dki_dcache_wb**(D3X) and **dki_dcache_wbinval**(D3X) functions, allow drivers to manage the data cache for DMA buffers or other purposes.

Return Values

None

See Also

The "Data Cache Write Back and Invalidation" section of the *IRIX Device Driver Programming Guide*

Note

The **dki_dcache_inval**, **dki_dcache_wb**(D3X), and **dki_dcache_wbinval**(D3X) functions replace **vflush**(D3X). On machines where a particular operation does not make sense, such as cache write back on a machine with a write through cache, the routine is provided as a stub routine which performs no operation. This allows drivers using these routines to work on all Silicon Graphics machines.

NAME

dki_dcache_wb – write back the data cache for a given range of virtual addresses

SYNOPSIS

#include "sys/types.h"

dki_dcache_wb(caddr_t *v_addr*, **unsigned** *len***);**

Arguments

v_addr	Can be either a user or kernel virtual address. If *v_addr* is a user virtual address, it is assumed to be that of the current mapped process. If, however, *v_addr* is a *k1seg* address, or if it is a user virtual or *k2seg* address and the page table entry specifies that the page is not cacheable, no operation is performed on the data cache for that page.
len	Gives the number of bytes over which to perform the operation.

DESCRIPTION

dki_dcache_wb writes back the data cache starting at v_addr address. This function, along with the **dki_dcache_inval**(D3X) and **dki_dcache_wbinval**(D3X) functions provide a sufficient set of functions to allow drivers to manage the data cache for DMA buffers or other purposes.

Return Values

None

See Also

The "Data Cache Write Back and Invalidation" section of the *IRIX Device Driver Programming Guide*

Note

The **dki_dcache_wb**, **dki_dcache_inval**(D3X), and **dki_dcache_wbinval**(D3X) functions replace **vflush**(D3X). On machines where a particular operation does not make sense, such as cache write back on a machine with a write through cache, the routine is provided as a stub routine which performs no operation. This allows drivers using these routines to work on all Silicon Graphics machines.

NAME

dki_dcache_wbinval – write back and invalidate the data cache for a given range of virtual addresses

SYNOPSIS

```
#include "sys/types.h"

dki_dcache_wbinval(caddr_t v_addr, unsigned len);
```

Arguments

len	Gives the number of bytes over which to perform the operation.
v_addr	Can be either a user or kernel virtual address. If *v_addr* is a user virtual address, it is assumed to be that of the current mapped process. If, however, *v_addr* is a *k1seg* address, or if it is a user virtual or *k2seg* address and the page table entry specifies that the page is not cacheable, no operation is performed on the data cache for that page.

DESCRIPTION

dki_dcache_wbinval writes back and invalidates the data cache starting at *v_addr* address. This function, along with the **dki_dcache_wb** and **dki_dcache_inval** functions provide a sufficient set of functions to allow drivers to manage the data cache for DMA buffers or other purposes.

Return Values

None

See Also

The "Data Cache Write Back and Invalidation" section of the *IRIX Device Driver Programming Guide*

Note

The **dki_dcache_inval**, **dki_dcache_wb**, and **dki_dcache_wbinval** functions replace **vflush**(D3X). On machines where a particular operation does not make sense, such as cache write back on a machine with a write through cache, the routine is provided as a stub routine which performs no operation. This allows drivers using these routines to work on all Silicon Graphics machines.

NAME

dma_map – load DMA mapping registers for an imminent transfer

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"
#include "sys/dmamap.h"

dma_map(dmamap_t *dmamap, caddr_t kernel_vaddr, int num_bytes);
```

DESCRIPTION

dma_map attempts to map *num_bytes* of main memory starting at the kernel virtual address *kernel_vaddr*, using the previously allocated DMA map *dmamap*. **dma_map** determines the actual physical memory locations for the given address and range and loads them into the mapping registers corresponding to the map. These mapping registers remain undisturbed until another call to **dma_map**.

Return Values

dma_map returns the actual number of bytes mapped. This number may be less than that requested if the number of map registers required exceeds the size of the given DMA map. 0 is returned if the arguments are invalid, for instance, if a *kernel_vaddr* is not word aligned.

See Also

dma_mapaddr(D3X), **dma_mapalloc**(D3X), **dma_mapfree**(D3X), **vme_adapter**(D3X)

NAME

dma_mapaddr – return the "bus virtual" address for a given map and address

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"
#include "sys/dmamap.h"

unsigned int dma_mapaddr(dmamap_t *dmamap, caddr_t kernel_vaddr);
```

DESCRIPTION

dma_mapaddr returns the bus virtual address corresponding to the given DMA map and kernel virtual address. This is the address that you should give to the device as the beginning "physical" address of the transfer. Before using **dma_mapaddr**, you should make a call to **dma_map**, to load the DMA mapping registers.

Return Values

The bus virtual address described above.

See Also

dma_map(D3X), **dma_mapalloc**(D3X), **dma_mapfree**(D3X), **vme_adapter**(D3X)

NAME

dma_mapalloc – allocate a DMA map

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"
#include "sys/dmamap.h"

dmamap_t *dma_mapalloc(int type, int adapter, int num_pages, int flags);
```

Arguments

type	Must be either *DMA_A32VME* or *DMA_A24VME* depending on the transfer desired. (The *DMA_SCSI* type is reserved for exclusive use by the SCSI host adapter driver.)
adapter	Specifies the I/O adapter to use, and should always be 0.
num_pages	Specifies the maximum number of mapping registers to allocate. Alternatively, you can think of *num_pages* as the maximum number of 4096 byte pages per transfer. You need to allocate an extra page for non-page aligned transfers-for example, a transfer of 4096 bytes starting at a non-aligned address actually requires two mapping registers.
flags	Reserved for future development. For now, you should always set it to 0.

DESCRIPTION

dma_mapalloc allocates DMA mapping registers on multiprocessor models and returns a pointer to a structure, of type *dmamap_t*, for later use by the mapping routine, **dma_map**. You need DMA maps to access main memory through VME A24 space. In addition, because DMA maps give you the ability to perform transfers to non-contiguous physical memory, you also want them for A32 access.

Use **dma_mapfree** to free the DMA mapping registers and other resources associated with a given map.

To determine which VME adapter a device is connected to, use the **vme_adapter**(D3X) function, where +k is the base address of the VME device, usually specified on the vector line of the device in the *master.d/* system file.

This call can block (it calls **psema**) if no maps are available, so it must never be called at interrupt time.

Return Values

dma_mapalloc returns a pointer to the DMA map structure on models that support DMA maps. On other models, **dma_mapalloc** returns –1 to indicate that DMA mapping is not possible on that model.

See Also

dma_map(D3X), **dma_mapaddr**(D3X), **vme_adapter**(D3X), **dma_mapfree**(D3X)

NAME

dma_mapfree – free a DMA map

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"
#include "sys/dmamap.h"

dma_mapfree(dmamap_t *dmamap);
```

DESCRIPTION

dma_mapfree frees the DMA mapping registers and other resources associated with a given map. To determine which VME adapter a device is connected to, use the **vme_adapter**(D3X) function, where +k is the base address of the VME device, usually specified on the vector line of the device in the *master.d/* system file.

This call can block (it calls **psema**(D3X)) if no maps are available, so it must never be called at interrupt time.

Return Values

The returned value of **dma_mapfree** conveys no useful information.

See Also

dma_map(D3X), **dma_mapaddr**(D3X), **dma_mapalloc**(D3X), **vme_adapter**(D3X)

NAME

drv_getparm – retrieve kernel state information

SYNOPSIS

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

int drv_getparm(ulong_t parm, ulong_t *value_p);
```

Arguments

parm	The kernel parameter to be obtained.
value_p	A pointer to the data space into which the value of the parameter is to be copied.

DESCRIPTION

drv_getparm returns the value of the parameter specified by *parm* in the location pointed to by *value_p*.

Return Values

On success, **drv_getparm** returns 0. On failure it returns –1 to indicate that *parm* specified an invalid parameter.

USAGE

drv_getparm does not explicitly check to see whether the driver has the appropriate context when the function is called. It is the responsibility of the driver to use this function only when it is appropriate to do so and to correctly declare the data space needed.

Valid values for *parm* are:

LBOLT Read the number of clock ticks since the last system reboot. The difference between the values returned from successive calls to retrieve this parameter provides an indication of the elapsed time between the calls in units of clock ticks. The length of a clock tick can vary across different implementations, and therefore drivers should not include any hard-coded assumptions about the length of a tick. The **drv_hztousec**(D3) and **drv_usectohz**(D3) functions can be used, as necessary, to convert between clock ticks and microseconds (implementation independent units).

TIME Read the time in seconds. This is the same time value that is returned by the **time**(2) system call. The value is defined as the time in seconds since 00:00:00 GMT, January 1, 1970. This definition presupposes that the administrator has set the correct system date and time.

UPROCP Retrieve a pointer to the process structure for the current process. The value returned in **value_p* is of type **(proc_t *)** and the only valid use of the value is as an argument to **vtop**(D3), or when calling **psignal**(D3) on those systems which do not have the new **proc_signal**(D3) interfaces. Since this value is associated with the current process, the caller must have process context (that is, must be at base level) when attempting to retrieve this value. Also, this value should only be used in the context of the process in which it was retrieved.

`UCRED`	Retrieve a pointer to the credential structure describing the current user credentials for the current process. The value returned in **value_p* is of type **`(cred_t *)`** and the only valid use of the value is as an argument to **`drv_priv`**(D3). Since this value is associated with the current process, the caller must have process context (that is, must be at base level) when attempting to retrieve this value. Also, this value should only be used in the context of the process in which it was retrieved.
`PGRP`	Read the process group identification number. This number determines which processes should receive a HANGUP or BREAK signal when detected by a driver.
`PPID`	Read process identification number.
`PSID`	Read process identification number.

Level

Base only when using the **`UPROCP`** or **`UCREDP argument values.`**
`Initialization, Base, or Interrupt when using the LBOLT or TIME argument values.`

Synchronization Constraints

Does not sleep.
Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function

REFERENCES

`drv_hztousec`(D3), **`drv_priv`**(D3), **`drv_usectohz`**(D3)

NAME

drv_hztousec – convert clock ticks to microseconds

SYNOPSIS

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

clock_t drv_hztousec(clock_t ticks);
```

Arguments

ticks	The number of clock ticks to convert to equivalent microseconds.

DESCRIPTION

drv_hztousec converts the length of time expressed by *ticks*, which is in units of clock ticks, into units of microseconds.

Return Values

drv_hztousec returns the number of microseconds equivalent to the *ticks* argument. No error value is returned. If the microsecond equivalent to *ticks* is too large to be represented as a **clock_t**, then the maximum **clock_t** value is returned.

USAGE

Several functions either take time values expressed in clock ticks as arguments [**itimeout**(D3), **delay**(D3)] or return time values expressed in clock ticks [**drv_getparm**(D3)]. The length of a clock tick can vary across different implementations, and therefore drivers should not include any hard-coded assumptions about the length of a tick. **drv_hztousec** and the complementary function **drv_usectohz**(D3) can be used, as necessary, to convert between clock ticks and microseconds.

Note that the time value returned by **drv_getparm** with an **LBOLT** argument will frequently be too large to represent in microseconds as a **clock_t**. When using **drv_getparm** together with **drv_hztousec** to time operations, drivers can help avoid overflow by converting the difference between return values from successive calls to **drv_getparm** instead of trying to convert the return values themselves.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

delay(D3), **drv_getparm**(D3), **drv_usectohz**(D3), **itimeout**(D3)

NAME

`drv_priv` – determine whether credentials are privileged

SYNOPSIS

`int drv_priv(cred_t *`*crp***`);`**

Arguments

crp	Pointer to the user credential structure.

DESCRIPTION

The **`drv_priv`** function determines whether the credentials specified by the credential structure pointed to by *crp* identify a privileged process.

Return Values

`drv_prv` returns 0 if the specified credentials identify a privileged process and **`EPERM`** otherwise.

USAGE

This function should only be used when file access modes and special minor device numbers are insufficient to provide the necessary protection for the driver operation being performed. Calls to **`drv_priv`** should replace all calls to **`suser`** and any explicit checks for effective user ID equal to zero in driver code.

A credential structure pointer is passed into various driver entry point functions [**`open`**(D2), **`close`**(D2), **`read`**(D2), **`write`**(D2), and **`ioctl`**(D2)] and can also be obtained by calling **`drv_getparm`**(D3) from base level driver code.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Warnings

The only valid use for a credential structure pointer is as an argument to **`drv_priv`**. The contents of a credential structure are not defined by the DDI / DKI and a driver may not examine the contents of the structure directly.

NAME

drv_setparm – set kernel state information

SYNOPSIS

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

int drv_setparm(ulong_t parm, ulong_t value);
```

Arguments

parm	The kernel parameter to be updated.
value	The value to be added to the parameter.

DESCRIPTION

drv_setparm verifies that *parm* corresponds to a kernel parameter that may be modified. If the value of *parm* corresponds to a parameter that may not be modified, –1 is returned. Otherwise, the parameter is incremented by *value*.

Return Values

If the function is successful, 0 is returned. Otherwise, –1 is returned to indicate that *parm* specified an invalid parameter.

USAGE

No checking is performed to determine the validity of *value*. It is the driver's responsibility to guarantee the correctness of *value*.

Valid values for *parm* are:

SYSCANC	Add *value* to **sysinfo.canch**. **sysinfo.canch** is a count of the number of characters received from a terminal device after the characters have been processed to remove special characters such as *break* or *backspace*.
SYSMINT	Add *value* to **sysinfo.mdmint**. **sysinfo.mdmint** is a count of the number of modem interrupts received.
SYSOUTC	Add *value* to **sysinfo.outch**. **sysinfo.outch** is a count of the number of characters output to a terminal device.
SYSRAWC	Add *value* to **sysinfo.rawc**. **sysinfo.rawc** is a count of the number of characters received from a terminal device, before canonical processing has occurred.
SYSRINT	Add *value* to **sysinfo.rcvint**. **sysinfo.rcvint** is a count of the number of interrupts generated by data ready to be received from a terminal device.
SYSXINT	Add *value* to **sysinfo.xmtint**. **sysinfo.xmtint** is a count of the number of interrupts generated by data ready to be transmitted to a terminal device.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

drv_getparm(D3)

drv_usectohz(D3)

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 to equivalent clock ticks.

DESCRIPTION

drv_usectohz converts the length of time expressed by *microsecs*, which is in units of microseconds, into units of clock ticks.

Return Values

The value returned is the smallest number of clock ticks that represent a time interval equal to or greater than the *microsecs* argument. No error value is returned. If the number of ticks equivalent to the *microsecs* argument is too large to be represented as a **clock_t**, then the maximum **clock_t** value will be returned.

USAGE

Several functions either take time values expressed in clock ticks as arguments [**itimeout**(D3), **delay**(D3)] or return time values expressed in clock ticks [**drv_getparm**(D3)]. The length of a clock tick can vary across different implementations, and therefore drivers should not include any hard-coded assumptions about the length of a tick. **drv_usectohz** and the complementary function **drv_hztousec**(D3) can be used, as necessary, to convert between microseconds and clock ticks.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

delay(D3), **drv_getparm**(D3), **drv_hztousec**(D3), **itimeout**(D3)

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.

DESCRIPTION

drv_usecwait causes the caller to busy-wait for at least the number of microseconds specified by *microsecs*. The amount of time spent busy-waiting may be greater than the time specified by *microsecs* but will not be less.

Return Values

None

USAGE

drv_usecwait should only be used to wait for short periods of time (less than a clock tick) or when it is necessary to wait without sleeping (for example, at interrupt level). When the desired delay is at least as long as clock tick and it is possible to sleep, the **delay**(D3) function should be used instead since it will not waste processor time busy-waiting as **drv_usecwait** does.

Because excessive busy-waiting is wasteful the driver should only make calls to **drv_usecwait** as needed, and only for as much time as needed. **drv_usecwait** does not raise the interrupt priority level; if the driver wishes to block interrupts for the duration of the wait, it is the driver's responsibility to set the priority level before the call and restore it to its original value afterward.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Warnings

Busy-waiting can increase the preemption latency experienced by high priority processes. Since short and bounded preemption latency can be critical in a real time environment, drivers intended for use in such an environment should not use this interface or should limit the length of the wait to an appropriately short length of time.

REFERENCES

delay(D3), **drv_hztousec**(D3), **drv_usectohz**(D3), **itimeout**(D3), **untimeout**(D3)

NAME

`dtimeout` – execute a function on a specified processor after a specified length of time

SYNOPSIS

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

toid_t dtimeout(void (*fn)(), void *arg, long ticks, pl_t pl,
      processorid_t processor, arg2, arg3, arg4);
```

Arguments

fn	Function to execute on the specified processor when the time increment expires.
arg, arg2, arg3, arg4	Argument to the function.
ticks	Number of clock ticks to wait before the function is called.
pl	The interrupt priority level at which the function will be called.
processor	Processor on which the function must execute.

DESCRIPTION

`dtimeout` causes the function specified by *fn* to be called after the time interval specified by *ticks*, on the processor specified by *processor*, at the interrupt priority level specified by *pl*. *arg* will be passed as the only argument to function *fn*. The **`dtimeout`** call returns immediately without waiting for the specified function to execute.

Return Values

If the function specified by *fn* is successfully scheduled, **`dtimeout`** returns a non-zero identifier that can be passed to **`untimeout`** to cancel the request. If the function could not be scheduled on the specified processor, **`dtimeout`** returns a value of 0.

USAGE

This directed timeout capability provides a form of dynamic processor binding for driver code.

Drivers should be careful to cancel any pending **`dtimeout`** functions that access data structures before these structures are de-initialized or deallocated.

fn Argument

The function specified by *fn* must neither sleep, reference process context, nor lower the interrupt priority level below *pl*.

After the time interval has expired, *fn* only runs if the processor is at base level. Otherwise, *fn* is deferred until some time in the near future.

If **`dtimeout`** is called holding a lock that is contended for by *fn*, the caller must hold the lock at a processor level greater than the base processor level.

ticks Argument

The length of time before the function is called is not guaranteed to be exactly equal to the requested time, but will be at least *ticks–1* clock ticks in length.

A *ticks* argument of 0 has the same effect as a *ticks* argument of 1. Both will result in an approximate wait of between 0 and 1 tick (possibly longer).

pl Argument

pl must specify a priority level greater than or equal to *pltimeout*; thus, *plbase* cannot be used. See **LOCK_ALLOC**(D3) for a list of values for *pl*. Your driver should treat pl as an "opaque" and should not try to compare or do any operation

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

itimeout(D3), **LOCK_ALLOC**(D3), **untimeout**(D3)

NAME

dupb – duplicate a message block

SYNOPSIS

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

mblk_t *dupb(mblk_t *bp);
```

Arguments

bp	Pointer to the message block to be duplicated.

DESCRIPTION

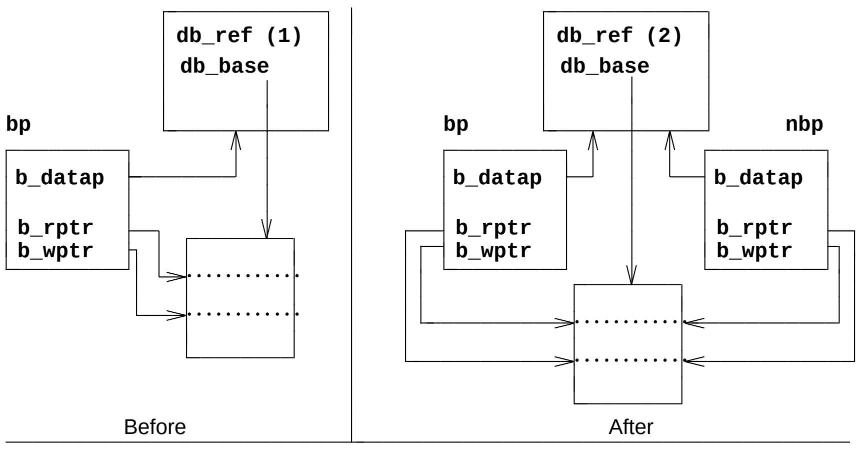
dupb creates a new message block structure that references the same data block that is referenced by *bp*. Unlike **copyb**(D3), **dupb** does not copy the information in the data block, but creates a new structure to point to it.

Return Values

On success, **dupb** returns a pointer to the new message block. On failure, it returns a **NULL** pointer.

USAGE

The following figure shows how the **db_ref** field of the data block structure has been changed from 1 to 2, reflecting the increase in the number of references to the data block. The new message block 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);

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

copyb(D3), **dupmsg**(D3), **datab**(D4), **msgb**(D4)

NAME

dupmsg – duplicate a message

SYNOPSIS

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

mblk_t *dupmsg(mblk_t *mp);
```

Arguments

mp	Pointer to the message.

DESCRIPTION

dupmsg forms a new message by duplicating the message blocks in the message pointed to by *mp* and linking them via their **b_cont** pointers.

Return Values

On success, **dupmsg** returns a pointer to the new message. On failure, it returns a **NULL** pointer.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See the **copyb**(D3) manual page for an example of **dupmsg**.

REFERENCES

copyb(D3), **copymsg**(D3), **dupb**(D3), **datab**(D4), **msgb**(D4)

NAME

eisa_dma_disable – disable recognition of hardware requests on a DMA channel

SYNOPSIS

#include <sys/eisa.h>

void eisa_dma_disable(vint_t *adap***, init** *chan***);**

Arguments

adap	Bus adapter number (zero on an Indigo2).
chan	Channel to be disabled.

DESCRIPTION

The **eisa_dma_disable** routine disables recognition of hardware requests on the DMA channel *chan*. The channel is then released and made available for other use.

The caller must ensure that it is acting on behalf of the channel owner, and that it makes sense to release the channel. The caller must ensure that the channel is in use for hardware-initiated DMA transfers and not software-initiated transfers.

Return Values

None

Level

Base or Interrupt

Notes

Does not sleep

eisa_dma_enable(D3X)

NAME

eisa_dma_enable – enable recognition of hardware requests on a DMA channel

SYNOPSIS

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

void eisa_dma_enable(vint_t adap, init chan);
```

Arguments

adap	Bus adapter number (zero on an Indigo2).
chan	Channel to be enabled.

DESCRIPTION

The **eisa_dma_enable** routine enables recognition of hardware requests on the DMA channel *chan*.

After enabling the channel for a hardware initiated transfer, this function calls the procedure **proc**() from the command block used to program the DMA hardware start sequence. It will then sleep awaiting completion of the dma operation specified by the command block, depending on the value of *mode*. Note that *mode* must be *EISA_DMA_NOSLEEP* in Release 5.1.

The caller must ensure that it is acting on behalf of the channel owner, and that it makes sense to release the channel. The caller must also ensure that the channel is in use for hardware-initiated DMA transfers and not software-initiated transfers.

Return Values

None

Level

Base or Interrupt

Notes

Does not sleep

NAME

`eisa_dma_free_buf` – free a previously allocated DMA buffer descriptor

SYNOPSIS

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

void eisa_dma_free_buf(struct eisa_dma_buf *dmabufptr);
```

Arguments

dmabufptr

Address of the allocated DMA buffer descriptor to be returned.

DESCRIPTION

`eisa_dma_free_buf` frees a DMA buffer descriptor. The *dmabufptr* argument must specify the address of a DMA buffer descriptor previously allocated by **`eisa_dma_get_buf`**().

Return Values

None

Level

Base or Interrupt

Notes

Does not sleep

NAME

eisa_dma_free_cb – free a previously allocated DMA command block

SYNOPSIS

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

void eisa_dma_free_cb(struct dma_cb *dmacbptr);
```

Arguments

dmacbptr

Address of the allocated DMA command block to be returned.

DESCRIPTION

eisa_dma_free_cb frees a DMA command block. The *dmacbptr* argument must specify the address of a DMA command block previously allocated by **eisa_dma_get_cb**().

Return Values

None

Level

Base or Interrupt

Notes

Does not sleep

NAME

`eisa_dma_get_buf` – allocated DMA buffer descriptor

SYNOPSIS

```
#include <sys/types.h>
#include <sys/eisa.h>

struct eisa_dma_buf *eisa_dma_get_buf(uchar_t mode);
```

Arguments

mode Specifies whether the caller is willing to sleep waiting for memory. If mode is set to *EISA_DMA_SLEEP*, the caller will sleep if necessary until the memory for a **`dma_buf`**() is available. If mode is set to *EISA_DMA_NOSLEEP*, the caller will not sleep, but **`eisa_dma_get_buf`** will return *NULL* if memory for a **`dma_buf`**(D4X) is not immediately available.

DESCRIPTION

`eisa_dma_get_buf` allocates memory for a DMA command block structure (see **`eisa_dma_buf`**(), zeros it out, and returns a pointer to the structure.

Return Values

`eisa_dma_get_buf` returns a pointer to the allocated DMA control block. If *EISA_DMA_NOSLEEP* is specified and memory for a **`eisa_dma_buf`**() is not immediately available, **`eisa_dma_get_buf`** returns a *NULL* pointer.

Level

Base only if mode is set to *EISA_DMA_SLEEP*. Base or Interrupt if mode is set to *EISA_DMA_NOSLEEP*.

Notes

Can sleep if mode is set to *DMA_SLEEP*.

NAME

eisa_dma_get_cb – allocated a DMA command block

SYNOPSIS

```
#include <sys/types.h>
#include <sys/eisa.h>

struct dma_cb *eisa_dma_get_cb(uchar_t mode);
```

Arguments

mode Specifies whether the caller is willing to sleep waiting for memory. If *mode* is set to *EISA_DMA_SLEEP*, the caller will sleep if necessary until the memory for a **eisa_dma_cb**() is available. If mode is set to *EISA_DMA_NOSLEEP*, the caller will not sleep, but **eisa_dma_get_cb** will return *NULL* if memory for a **eisa_dma_buf**() is not immediately available.

DESCRIPTION

eisa_dma_get_cb allocates memory for a DMA command block structure (see **eisa_dma_cb**(), zeros it out, and returns a pointer to the structure.

Return Values

eisa_dma_get_cb returns a pointer to the allocated DMA control block. If *EISA_DMA_NOSLEEP* is specified and memory for a **eisa_dma_cb**() is not immediately available, **eisa_dma_get_cb** returns a *NULL* pointer.

Level

Base only if mode is set to *EISA_DMA_SLEEP*. Base or Interrupt if mode is set to *EISA_DMA_NOSLEEP*.

Notes

Can sleep if mode is set to *EISA_DMA_SLEEP*.

NAME

`eisa_dma_prog` – program a DMA operation for a subsequent software request

SYNOPSIS

`#include <sys/eisa.h>`

`void eisa_dma_prog(vint_t` *adap*, **`struct eisa_dma_cb *`***dmacbptr*, **`init`** *chan*, **`uchar_t`** *mode*

Arguments

adap Bus adapter number (zero on an Indigo2).

dmacbptr
Pointer to the DMA command block specifying the DMA operation.

chan Channel over which the DMA operation is to take place.

mode Specifies whether the caller is willing to sleep waiting for to allocate the desired DMA channel. If mode is set to *EISA_DMA_NOSLEEP*, then the caller will sleep if necessary until the requested channel becomes available for its use. If *mode* is set to *EISA_DMA_SLEEP*, then the caller will not sleep, but **`eisa_dma_prog`** will return FALSE if the requested DMA channel is not immediately available.

DESCRIPTION

The **`eisa_dma_prog`** routine programs the DMA channel *chan* for the operation specified by the DMA command block whose address is given by *dmacbptr*. Note that **`eisa_dma_prog`** does not initiate the DMA transfer. Instead, the transfer will be initiated by a subsequent request initiated by **`eisa_dma_swstart`**() or **`eisa_dma_enable`**().

To program the operation, **`eisa_dma_prog`** requires exclusive use of the specified DMA channel. The caller may specify, via the mode argument, whether **`eisa_dma_prog`** should sleep waiting for a busy channel to become available. If the specified channel is in use and mode is set to *EISA_DMA_SLEEP*, then **`eisa_dma_prog`** will sleep until the channel becomes available for its use. Otherwise, if *EISA_DMA_NOSLEEP* is specified and the requested channel is not immediately available, **`eisa_d ma_prog`** will not program the channel, but will simply return a value of *FALSE*.

Return Values

`eisa_dma_prog` returns the value *TRUE* on success and *FALSE* otherwise.

Level

Base only if either *mode* is set to *EISA_DMA_SLEEP*.

Notes

Can sleep if *mode* is set to *DMA_SLEEP* or the routine specified by the *proc* field of the **`eisa_dma_cb`** structure sleeps.

NAME

eisa_dma_stop – stop software-initiated DMA operation on a channel and release it

SYNOPSIS

#include <sys/eisa.h>

void eisa_dma_stop(vint_t *adap***, init** *chan***);**

Arguments

adap	Bus adapter number (zero on an Indigo2).
chan	Channel on which the DMA operation is to be stopped.

DESCRIPTION

eisa_dma_stop stops a software-initiated DMA operation in progress on the channel *chan*. The channel is then released and made available for other use.

The caller must ensure that it is acting on behalf of the channel owner, and that it makes sense to release the channel. The caller must also ensure that the channel is in use for software-initiated DMA transfers and not hardware-initiated transfers.

Return Values

None

Level

Base or Interrupt

Notes

Does not sleep.

NAME

eisa_dma_swstart – initiate a DMA operation via software request

SYNOPSIS

#include <sys/eisa.h>

void eisa_dma_swstart(struct dma_cb ****dmacbptr*, **init *chan*, **uchar_t** *mode***);**

Arguments

dmacbptr
: Address of the allocated DMA command block to be returned.

chan
: Channel over which the DMA operation is to take place.

mode
: Specifies whether the caller is willing to sleep waiting for the operation to complete. If mode is set to *EISA_DMA_NOSLEEP*, then **eisa_dma_swstart** starts the operation but does not wait for the operation to complete and instead returns to the caller immediately. If *mode* is set to *EISA_DMA_SLEEP*, then **eisa_dma_swstart** starts the operation and then waits for the operation to complete, and returns to the caller after the operation has finished.

DESCRIPTION

The **eisa_dma_swstart** routine initiates a DMA operation previously programmed by **eisa_dma_prog**(). If *mode* is set to *DMA_SLEEP*, then **eisa_dma_swstart** returns to the caller after the operation completes. If mode is set to *EISA_DMA_NOSLEEP*, then **eisa_dma_swstart** returns to the caller immediately after starting the operation.

Return Values

None

Level

Base only if mode is set to *EISA_DMA_SLEEP*. Base or Interrupt if mode is set to *EISA_DMA_NOSLEEP*.

Notes

The operation being initiated must have already been programmed on the specified channel by **eisa_dma_prog**().

Will sleep if *mode* is set to *EISA_DMA_SLEEP*; *mode* must be *EISA_DMA_NOSLEEP* in Release 5.1.

NAME

enableok – allow a queue to be serviced

SYNOPSIS

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

void enableok(queue_t *q);
```

Arguments

q	Pointer to the queue.

DESCRIPTION

The **enableok** function allows the service routine of the queue pointed to by *q* to be rescheduled for service. It cancels the effect of a previous use of the **noenable**(D3) function on *q*.

Return Values

None

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

The caller cannot have the stream frozen [see **freezestr**(D3)] when calling this function.

Examples

The **qrestart** routine uses two STREAMS functions to re-enable a queue that has been disabled. The **enableok** function removes the restriction that prevented the queue from being scheduled when a message was enqueued. Then, if there are messages on the queue, it is scheduled by calling **qenable**(D3).

```
void
qrestart(q)
   queue_t *q;
{
   enableok(q);
   if (q->q_first)
         qenable(q);
}
```

REFERENCES

noenable(D3), **qenable**(D3), **queue**(D4), **srv**(D2)

NAME

`esballoc` – allocate a message block using an externally-supplied buffer

SYNOPSIS

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

mblk_t *esballoc(uchar_t *base, int size, int pri, frtn_t *fr_rtnp);
```

Arguments

base	Address of driver-supplied data buffer.
size	Number of bytes in data buffer.
pri	Priority of allocation request (used to allocate the message and data blocks).
fr_rtnp	Pointer to the free-routine data structure.

DESCRIPTION

`esballoc` creates a STREAMS message and attaches a driver-supplied data buffer in place of a STREAMS data buffer. It allocates a message and data block header only. The driver-supplied data buffer, pointed to by *base*, is used as the data buffer for the message.

When **`freeb`**(D3) is called to free the message, on the last reference to the message, the driver's free-routine, specified by the **`free_func`** field in the **`free_rtn`**(D4) structure, is called with one argument, specified by the **`free_arg`** field, to free the data buffer.

Return Values

On success, a pointer to the newly allocated message block is returned. On failure, **`NULL`** is returned.

USAGE

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.

When the **`free_func`** function runs, interrupts from all STREAMS devices will be blocked. It has no user context and may not call any routine that sleeps. The function may not access any dynamically allocated data structures that might no longer exist when it runs.

The *pri* argument is no longer used, but is retained for compatibility. Some implementations may choose to ignore this argument.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

allocb(D3), **esbbcall**(D3), **freeb**(D3), **free_rtn**(D4)

NAME

esbbcall – call a function when an externally-supplied buffer can be allocated

SYNOPSIS

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

toid_t esbbcall(int pri, void (*func)(), long arg);
```

Arguments

pri	Priority of the **esballoc**(D3) allocation request.
func	Function to be called when a buffer becomes available.
arg	Argument to the function to be called when a buffer becomes available.

DESCRIPTION

If **esballoc**(D3) is unable to allocate a message block header and a data block header to go with its externally supplied data buffer, the function **esbbcall** can be used to schedule the routine *func*, to be called with the argument *arg* when memory becomes available. **esbbcall**, like **bufcall**(D3), serves, in effect, as a timeout call of indeterminate length.

Return Values

On success, **esbbcall** returns a non-zero value that identifies the scheduling request. On failure, **esbbcall** returns 0.

USAGE

When *func* runs, all interrupts from STREAMS devices will be blocked. On multiprocessor systems, the interrupts will be blocked only on the processor on which *func* is running. *func* will have no user context and may not call any function that sleeps.

Even when *func* is called, **esballoc** can still fail if another module or driver had allocated the memory before *func* was able to call **allocb**.

The *pri* argument is no longer used, but is retained for compatibility.

The non-zero identifier returned by **esballoc** may be passed to **unbufcall**(D3) to cancel the request.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

allocb(D3), **bufcall**(D3), **esballoc**(D3), **itimeout**(D3), **unbufcall**(D3)

etoimajor(D3)

NAME

etoimajor – convert external to internal major device number

SYNOPSIS

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

int etoimajor(major_t emaj);
```

Arguments

emaj	External major number.

DESCRIPTION

etoimajor converts the external major number *emaj* to an internal major number.

Return Values

etoimajor returns the internal major number or **NODEV** if the external major number is invalid.

USAGE

See **getemajor**(D3) for a description of external and internal major numbers.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

getemajor(D3), **geteminor**(D3), **getmajor**(D3), **getminor**(D3), **itoemajor**(D3), **makedevice**(D3)

NAME

flushband – flush messages in a specified priority band

SYNOPSIS

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

void flushband(queue_t *q, uchar_t pri, int flag);
```

Arguments

q	Pointer to the queue.
pri	Priority band of messages to be flushed.
flag	Determines messages to flush.

DESCRIPTION

The **flushband** function 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*.

If the band's count falls below the low water mark and someone wants to write to the band, the nearest upstream or downstream service procedure is enabled.

Return Values

None

USAGE

Valid values for *flag* are:

FLUSHDATA	Flush only data messages (types **M_DATA**, **M_DELAY**, **M_PROTO**, and **M_PCPROTO**).
FLUSHALL	Flush all messages.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **put**(D2) for an example of **flushband**.

REFERENCES

flushq(D3), **put**(D2), **queue**(D4)

NAME

`flushbus` – make sure contents of the write buffer are flushed to the system bus

SYNOPSIS

`flushbus();`

DESCRIPTION

`flushbus` performs the necessary actions to ensure that any writes in the write buffer have actually been flushed to the system bus. This is sometimes necessary when a device requires delays between PIOs, particularly between a write and a read, since they might otherwise arrive at the device back-to-back.

NAME

`flushq` – flush messages on a queue

SYNOPSIS

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

void flushq(queue_t *q, int flag);
```

Arguments

q	Pointer to the queue to be flushed.
flag	Determines messages to flush.

DESCRIPTION

`flushq` frees messages on a queue by calling **`freemsg`**(D3) for each message. If the queue's count falls below the low water mark and someone wants to write to the queue, the nearest upstream or downstream service procedure is enabled.

Return Values

None

USAGE

Valid values for *flag* are:

`FLUSHDATA`	Flush only data messages (types **`M_DATA`**, **`M_DELAY`**, **`M_PROTO`**, and **`M_PCPROTO`**).
`FLUSHALL`	Flush all messages.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **`put`**(D2) for an example of **`flushq`**.

REFERENCES

`flushband`(D3), **`freemsg`**(D3), **`put`**(D2), **`putq`**(D3), **`queue`**(D4)

NAME

freeb – free a message block

SYNOPSIS

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

void freeb(mblk_t *bp);
```

Arguments

bp	Pointer to the message block to be deallocated.

DESCRIPTION

freeb deallocates a message block. If the reference count of the **db_ref** member of the **datab**(D4) structure is greater than 1, **freeb** decrements the count and returns. Otherwise, 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 **esballoc**(D3), the driver is notified that the attached data buffer needs to be freed by calling the free-routine [see **free_rtn**(D4)] associated with the data buffer. Once this is accomplished, **freeb** releases the STREAMS resources associated with the buffer.

Return Values

None

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **copyb**(D3) for an example of **freeb**.

REFERENCES

allocb(D3), **dupb**(D3), **esballoc**(D3), **datab**(D4), **free_rtn**(D4), **msgb**(D4)

NAME

`freemsg` – free a message

SYNOPSIS

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

void freemsg(mblk_t *mp);
```

Arguments

mp	Pointer to the message to be deallocated.

DESCRIPTION

`freemsg` frees all message blocks, data blocks, and data buffers associated with the message pointed to by *mp*. **`freemsg`** walks down the **`b_cont`** list [see **`msgb`**(D4)], calling **`freeb`**(D3) for every message block in the message.

Return Values

None

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **`copymsg`**(D3) for an example of **`freemsg`**.

REFERENCES

`freeb`(D3), **`msgb`**(D4)

NAME

freerbuf – free a raw buffer header

SYNOPSIS

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

void freerbuf(buf_t *bp);
```

Arguments

bp	Pointer to a previously allocated buffer header structure.

DESCRIPTION

freerbuf frees a raw buffer header previously allocated by **getrbuf**(D3).

Return Values

None

USAGE

freerbuf may not be used on a buffer header obtained through an interface other than **getrbuf**.

freerbuf is typically called from a driver's **biodone** (D3) routine, as specified in the **b_iodone** field of the **buf**(D4) structure.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

biodone(D3), **bioreset**(D3), **biowait**(D3), **buf**(D4), **getrbuf**(D3)

NAME

freesema – free the resources associated with a semaphore

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"

freesema(sema_t *semap);
```

DESCRIPTION

freesema frees all resources associated with a semaphore. Use freesema to free dynamically allocated semaphores that are no longer needed. If the semaphores are part of a dynamically allocated structure, you must use *freesema* to free the semaphores before you free the structure containing the semaphores.

For **freesema**, *semap* is a pointer to the semaphore you want to deallocate.

Return Values

None

See Also

cpsema(D3X), **cvsema**(D3X), **psema**(D3X), **vsema**(D3X), **sleep**(D3), **vpsema**(D3X), **SLEEP_ALLOC**(D3), **SLEEP_DEALLOC**(D3)

NAME

fubyte – fetch (read) a byte from user space

SYNOPSIS

int fubyte(char ***_usr_v_addr_);**

DESCRIPTION

fubyte reads a single (8-bit) byte from the specified address, 2, in the currently mapped user process address space.

Return Values

Upon successful completion, **fubyte** returns the value of the byte at 2, a value from 0 to 255.

Otherwise, **fubyte** returns –1, indicating an invalid user virtual address.

See Also

fuword(D3X), **subyte**(D3X), **copyin**(D3)

NAME

fuword – fetch (read) a word from user space

SYNOPSIS

int fuword(int ***_usr_v_addr_);**

DESCRIPTION

fuword reads a (32-bit) word in the currently mapped user process' address space. Use *user_v_addr*, to specify the word you want to read.

Return Values

Upon successful completion, **fuword** returns the value from the requested location. Otherwise, **fuword** returns –1, indicating an invalid user virtual address.

See Also

fubyte(D3X), **suword**(D3X), **copyin**(D3)

NAME

geteblk – get an empty buffer

SYNOPSIS

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

buf_t *geteblk(void);
```

DESCRIPTION

geteblk retrieves a buffer [see **buf**(D4)] from the buffer cache and returns a pointer to the buffer header. If a buffer is not available, **geteblk** sleeps until one is available.

Return Values

A pointer to the buffer header structure is returned.

USAGE

When the driver **strategy**(D2) routine receives a buffer header from the kernel, all the necessary members are already initialized. However, when a driver allocates buffers for its own use, it must set up some of the members before calling its **strategy** routine.

The following list describes the state of these members when the buffer header is received from **geteblk**:

b_flags	is set to indicate the transfer is from the user's buffer to the kernel. The driver must set the **B_READ** flag if the transfer is from the kernel to the user's buffer.
b_edev	is set to **NODEV** and must be initialized by the driver.
b_bcount	is set to 1024.
b_un.b_addr	is set to the buffer's virtual address.
b_blkno	is not initialized by **geteblk**, and must be initialized by the driver

Typically, block drivers do not allocate buffers. The buffer is allocated by the kernel, and the associated buffer header is used as an argument to the driver **strategy** routine. However, to implement some special features, such as **ioctl**(D2) commands that perform I/O, the driver may need its own buffer space. The driver can get the buffer space from the system by using **geteblk** or **ngeteblk**(D3). If the driver chooses to use its own memory for the buffer, it can allocate a buffer header only using **getrbuf**(D3).

Buffers allocated via **geteblk** must be freed using either **brelse**(D3) or **biodone**(D3).

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

REFERENCES

biodone(D3), **biowait**(D3), **brelse**(D3), **buf**(D4), **ngeteblk**(D3)

getemajor(D3)

NAME

getemajor – get external major device number

SYNOPSIS

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

major_t getemajor(dev_t dev);
```

Arguments

dev	External device number.

DESCRIPTION

getemajor returns the external major number given a device number, *dev*.

Return Values

The external major number.

USAGE

External major numbers are visible to the user. Internal major numbers are only visible in the kernel. Since, on some architectures, the range of major numbers may be large and sparsely populated, the kernel keeps a mapping between external and internal major numbers to save space.

All driver entry points are passed device numbers using external major numbers.

Usually, a driver with more than one external major number will have only one internal major number. However, some system implementations map one-to-one between external and internal major numbers. Here, the internal major number is the same as the external major number and the driver may have more than one internal major number.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

etoimajor(D3), **geteminor**(D3), **getmajor**(D3), **getminor**(D3), **makedevice**(D3)

NAME

geteminor – get external minor device number

SYNOPSIS

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

minor_t geteminor(dev_t dev);
```

Arguments

dev	External device number.

DESCRIPTION

geteminor returns the external minor number given a device number, *dev*.

Return Values

The external minor number.

USAGE

External minor numbers are visible to the user. Internal minor numbers are only visible in the kernel. Since, on some architectures, a driver can support more than one external major device that maps to the same internal major device, the kernel keeps a mapping between external minor numbers and internal minor numbers to allow drivers to index arrays more easily. For example, a driver may support two devices, each with five minor numbers. The user may see each set of minor numbers numbered from zero to four, but the driver sees them as one set of minor numbers numbered from zero to nine.

All driver entry points are passed device numbers using external minor numbers.

Systems that map external major device numbers one-to-one with internal major numbers also map external minor numbers one-to-one with internal minor numbers.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

etoimajor(D3), **getemajor**(D3), **getmajor**(D3), **getminor**(D3), **makedevice**(D3)

NAME

geterror – retrieve error number from a buffer header

SYNOPSIS

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

int geterror(struct buf_t *bp);
```

Arguments

bp	Pointer to the buffer header.

DESCRIPTION

geterror is called to retrieve the error number from the error field of a buffer header (**buf**(D4) structure).

Return Values

An error number indicating the error condition of the I/O request is returned. If the I/O request completed successfully, 0 is returned.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

REFERENCES

buf(D4), **errnos**(D5)

NAME

getmajor – get internal major device number

SYNOPSIS

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

major_t getmajor(dev_t dev);
```

Arguments

dev	Internal device number.

DESCRIPTION

The **getmajor** function extracts the internal major number from a device number.

Return Values

The internal major number.

USAGE

No validity checking is performed. If *dev* is invalid, an invalid number is returned.

See **getemajor**(D3) for an explanation of external and internal major numbers.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

etoimajor(D3), **getemajor**(D3), **geteminor**(D3), **getminor**(D3), **makedevice**(D3)

getminor(D3)

NAME

getminor – get internal minor device number

SYNOPSIS

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

minor_t getminor(dev_t dev);
```

Arguments

dev	Internal device number.

DESCRIPTION

The **getminor** function extracts the internal minor number from a device number.

Return Values

The internal minor number.

USAGE

No validity checking is performed. If *dev* is invalid, an invalid number is returned.

See **getemajor**(D3) for an explanation of external and internal major numbers.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

etoimajor(D3), **getemajor**(D3), **geteminor**(D3), **getmajor**(D3), **makedevice**(D3)

NAME

getnextpg – get next page pointer

SYNOPSIS

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

struct pfdat *getnextpg(buf_t (*bp)(), struct pfdat *pp);
```

Arguments

bp	Pointer to the buffer header structure
pp	Pointer to the previous pfdat structure returned.

DESCRIPTION

getnextpg will return a pointer to the next page (pfdat) in a buffer header's page list (see buf(D4)) during a paged-I/O request. A paged-I/O request is identified by the **B_PAGEIO** flag being set and the **B_MAPPED** flag being clear in the **b_flags** field of the buffer header passed to a driver's **strategy**(D2) routine.

Given a buffer header, **bp**, and a pointer to the page, **pp**, returned from the previous call to **getnextpg**, the next page is returned. If **pp** is NULL, the first page in the page list is returned.

Level

Base or Interrupt.

Notes

Does not sleep.

See Also

strategy(D2), **bp_mapin**(D3), **bp_mapout**(D3), **pptophys**(D3X), **buf**(D4)

NAME

getq – get the next message from a queue

SYNOPSIS

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

mblk_t *getq(queue_t *q);
```

Arguments

q	Pointer to the queue from which the message is to be retrieved.

DESCRIPTION

getq gets the next available message from the top of the queue pointed to by *q*. It handles flow control, restarting I/O that was blocked as needed.

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.

USAGE

getq is typically used by service routines [see **srv**(D2)] to retrieve queued messages.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

Examples

See **srv**(D2) for an example of **getq**.

REFERENCES

bcanput(D3), **canput**(D3), **putbq**(D3), **putq**(D3), **qenable**(D3), **rmvq**(D3), **srv**(D2)

NAME

getrbuf – get a raw buffer header

SYNOPSIS

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

buf_t *getrbuf(long flag);
```

Arguments

flag	Indicates whether the caller should sleep for free space.

DESCRIPTION

getrbuf allocates the space for a buffer header [see **buf**(D4)]. If *flag* is set to **KM_SLEEP**, the caller will sleep if necessary until memory is available. If *flag* is set to **KM_NOSLEEP**, the caller will not sleep, but **getrbuf** will return **NULL** if memory is not immediately available.

Return Values

Upon successful completion, **getrbuf** returns a pointer to the allocated buffer header. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **getrbuf** returns a **NULL** pointer.

USAGE

getrbuf is used when a block driver is performing raw I/O (character interface) and needs to set up a buffer header that is not associated with a system-provided data buffer. The driver provides its own memory for the data buffer.

After allocating the buffer header, the caller must set the **b_iodone** field to the address of an iodone handler to be invoked when the I/O is complete [see **biodone**(D3)]. The caller must also initialize the following fields:

b_flags	Must be modified to indicate the direction of data transfer. Initially, it is set to indicate the transfer is from the user's buffer to the kernel. The driver must set the **B_READ** flag if the transfer is from the kernel to the user's buffer.
b_edev	Must be initialized to the proper device number.
b_bcount	Must be set to the number of bytes to transfer.
b_un.b_addr	Must be set to the virtual address of the caller-supplied buffer.
b_blkno	Must be set to the block number to be accessed.
b_resid	Must be set to the same value as **b_bcount**.
b_bufsize	Can be used to remember the size of the data buffer associated with the buffer header.

Typically, block drivers do not allocate buffers. The buffer is allocated by the kernel and the associated buffer header is used as an argument to the driver **strategy** routine. However, to implement some special features, such as **ioctl**(D2) commands that perform I/O, the driver may need its own buffer space. The driver can get the buffer space from the system by using **geteblk**(D3) or **ngeteblk**(D3). Or the driver can choose to use its own memory for the buffer and only allocate a buffer header with

getrbuf.

Level

Base only if *flag* is set to **KM_SLEEP**.

Base or Interrupt if *flag* is set to **KM_NOSLEEP**.

Synchronization Constraints

May sleep if *flag* is set to **KM_SLEEP**.

Driver-defined basic locks and read/write locks may be held across calls to this function if *flag* is **KM_NOSLEEP**, but may not be held if *flag* is **KM_SLEEP**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

REFERENCES

biodone(D3), **bioreset**(D3), **biowait**(D3), **buf**(D4), **freerbuf**(D3)

NAME

hwcpin – copy data from device memory to main memory using 16-bit reads

SYNOPSIS

#include "sys/types.h"

hwcpin(unsigned short ***_from_, caddr_t** _to_**, int** _num_bytes_**);**

DESCRIPTION

hwcpin efficiently copies data from device memory to main memory using 16-bit reads only. Use **hwcpin** when transferring data from VME devices that understand only 16-bit halfwords.

Return Values

None

Note

hwcpin is similar to **bcopy**(D3) in that it does not verify the accessibility of the memory before attempting the transfer.

See Also

hwcpout(D3X)

hwcpout(D3X)

NAME

`hwcpout` – copy data from main memory to device memory using 16-bit writes

SYNOPSIS

`#include "sys/types.h"`

`hwcpout(caddr_t` from**`,`** **`unsigned short *`***to***`,`** **`int`** *num_bytes***`);`**

DESCRIPTION

`hwcpout` efficiently copies data from main memory to device memory using only 16-bit writes. Use **`hwcpout`** when transferring data to VME devices that understand only 16-bit halfwords.

Return Values

None

Note

`hwcpout` is similar to **`bcopy`**(D3) in that it does not verify the accessibility of the memory before attempting the transfer.

See Also

`hwcpin`(D3X)

NAME

`initnsema` – initialize a synchronizing semaphore to a given value

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"

initnsema(sema_t *semap, int val, char *name);
```

DESCRIPTION

`initnsema` initializes an IRIX synchronizing semaphore (a structure of type *sema_t*). Use synchronizing semaphores to synchronize multiple processes. You must allocate a semaphore before you can use it in a semaphore operation, such as **`psema`** or **`vsema`**. You can declare semaphores in line by using the *sema_t* type, or you can allocate them dynamically by using the kernel memory allocator, **`kern_malloc`**(D3X). In the case of an already allocated semaphore struct, initsema fills it.

The *val* parameter expects the initial value to which you want to set the semaphore. The *name* parameter of **`initnsema`** expects a pointer to an eight character string that contains the name you want to assign to the semaphore. This name may be used by debugging utilities.

Return Values

None

See Also

`initnsema_mutex`(D3X), **`cpsema`**(D3X), **`cvsema`**(D3X), **`psema`**(D3X), **`vsema`**(D3X), **`sleep`**(D3), **`vpsema`**(D3X), **`SLEEP_ALLOC`**(D3), **`SLEEP_DEALLOC`**(D3)

NAME

initnsema_mutex – initialize a mutex semaphore to one

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"

initnsema_mutex(sema_t *semap, char *name);
```

DESCRIPTION

initnsema_mutex initializes an IRIX mutual exclusion (mutex) semaphore (a structure of type *sema_t*). Use mutex semaphores to synchronize access to critical sections. You must allocate a semaphore before you can use it in a semaphore operation, such as **psema** or **vsema**. You can declare semaphores in line by using the *sema_t* type, or you can allocate them dynamically by using the kernel memory allocator, **kern_malloc**(D3X). In the case of an already allocated semaphore struct, initsema_mutex fills it.

The *name* parameter of **initnsema** expects a pointer to an eight character string that contains the name you want to assign to the semaphore. This name may be used by debugging utilities.

Return Values

None

See Also

initnsema(D3X), **cpsema**(D3X), **cvsema**(D3X), **psema**(D3X), **vsema**(D3X), **sleep**(D3), **vpsema**(D3X), **SLEEP_ALLOC**(D3), **SLEEP_DEALLOC**(D3)

NAME

insq – insert a message into a queue

SYNOPSIS

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

int insq(queue_t ***q*, mblk_t ***emp***, mblk_t ***nmp***);**

Arguments

q	Pointer to the queue containing message *emp*.
emp	Pointer to the existing message before which the new message is to be inserted.
nmp	Pointer to the new message to be inserted.

DESCRIPTION

insq inserts a message into a queue. The message to be inserted, *nmp*, is placed in the queue pointed to by *q*, immediately before the message, *emp*. If *emp* is **NULL**, the new message is placed at the end of the queue. All flow control parameters are updated. The service procedure is scheduled to run unless disabled by a previous call to **noenable**(D3).

Return Values

If *nmp* was successfully enqueued, **insq** returns 1. Otherwise, **insq** returns 0.

USAGE

Messages are ordered in the queue based on their priority, as described in **srv**(D2). If an attempt is made to insert a message out of order in the queue, then *nmp* is not enqueued.

The insertion can fail if there is not enough memory to allocate the accounting data structures used with messages whose priority bands are greater than zero.

If *emp* is non-**NULL**, it must point to a message in the queue pointed to by *q*, or a system panic could result.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Example

This routine illustrates the use of **insq** to insert a message into the middle of a queue. This routine can be used to strip all the **M_PROTO** headers off all messages on a queue. We traverse the list of messages on the queue, looking for **M_PROTO** messages (line 9). When one is found, we remove it from the queue using **rmvq**(D3) (line 10). If there is no data portion to the message (line 11), we free the entire message using **freemsg**(D3). Otherwise, for every **M_PROTO** message block in the message, we strip the **M_PROTO** block off using **unlinkb**(D3) (line 15) and free the message block using **freeb**(D3). When the header has been stripped, the data portion of the message is inserted back into the queue where it was originally found (line 19).

```
 1  void
 2  striproto(q)
 3     queue_t *q;
 4  {
 5     mblk_t *emp, *nmp, *mp;

 6     mp = q->q_first;
 7     while (mp) {
 8             emp = mp->b_next;
 9             if (mp->b_datap->db_type == M_PROTO) {
10                     rmvq(q, mp);
11                     if (msgdsize(mp) == 0) {
12                             freemsg(mp);
13                     } else {
14                             while (mp->b_datap->db_type == M_PROTO) {
15                                     nmp = unlinkb(mp);
16                                     freeb(mp);
17                                     mp = nmp;
18                             }
19                             insq(q, emp, mp);
20                     }
21             }
22             mp = emp;
23     }
24  }
```

REFERENCES

getq(D3), **putbq**(D3), **putq**(D3), **rmvq**(D3), **srv**(D2)

NAME

itimeout – execute a function after a specified length of time

SYNOPSIS

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

toid_t itimeout(void (*func)(), void *arg, long ticks,
                                 pl_t pl, arg2, arg3, arg4);
```

Arguments

func	Function to execute when the time increment expires.
arg, arg2, arg3, arg4	Argument to the function.
ticks	Number of clock ticks to wait before the function is called.
pl	The interrupt priority level at which the function will be called.

DESCRIPTION

itimeout causes the function specified by *func* to be called after the time interval specified by *ticks*, at the interrupt priority level specified by *pl*. *arg* will be passed as the only argument to function *func*. The **itimeout** call returns immediately without waiting for the specified function to execute.

Return Values

If the function specified by *func* is successfully scheduled, **itimeout** returns a non-zero identifier that can be passed to **untimeout** to cancel the request. If the function could not be scheduled, **itimeout** returns a value of 0.

USAGE

pl must specify a priority level greater than or equal to *pltimeout*; thus, *plbase* cannot be used. See **LOCK_ALLOC**(D3) for a list of values for *pl*. Your driver should treat *pl* as an "opaque" and should not try to compare or do any operation.

The length of time before the function is called is not guaranteed to be exactly equal to the requested time, but will be at least *ticks–1* clock ticks in length.

The function specified by *func* must neither sleep, reference process context, nor lower the interrupt priority level below *pl*.

After the time interval has expired, *func* only runs if the processor is at base level. Otherwise, *func* is deferred until some time in the near future.

If **itimeout** is called holding a lock that is contended for by *func*, the caller must hold the lock at a processor level greater than the base processor level.

A *ticks* argument of 0 has the same effect as a *ticks* argument of 1. Both will result in an approximate wait of between 0 and 1 tick (possibly longer).

Drivers should be careful to cancel any pending **`itimeout`** functions that access data structures before these structures are de-initialized or deallocated.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **`copyb`**(D3) for an example of **`itimeout`**.

REFERENCES

`dtimeout`(D3), **`LOCK_ALLOC`**(D3), **`untimeout`**(D3)

NAME

`itoemajor` – convert internal to external major device number

SYNOPSIS

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

int itoemajor(major_t imaj, int prevemaj);
```

Arguments

imaj	Internal major number.
prevemaj	Most recently obtained external major number (or **`NODEV`**, if this is the first time the function has been called).

DESCRIPTION

`itoemajor` converts the internal major number *imaj* to the external major number. The external-to-internal major number mapping can be many-to-one, and so any internal major number may correspond to more than one external major number.

Return Values

External major number, or **`NODEV`**, if all have been searched.

USAGE

By repeatedly invoking this function and passing the most recent external major number obtained, the driver can obtain all possible external major number values.

See **`getemajor`**(D3) for an explanation of external and internal major numbers.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`etoimajor`(D3), **`getemajor`**(D3), **`geteminor`**(D3), **`getmajor`**(D3), **`getminor`**(D3), **`makedevice`**(D3)

kern_calloc(D3X)

NAME

`kern_calloc` – allocate storage for objects of a specified size

SYNOPSIS

```
#include "sys/types.h"

caddr_t
kern_calloc(int n, int object_size);
```

DESCRIPTION

`kern_calloc` allocates, and zeroes, storage for *n* objects of size *object_size* bytes. If necessary, the function sleeps until the entire requested memory is available. Therefore, do not call this function from an interrupt routine. The allocated space is aligned for the given object size.

Return Values

`kern_calloc` returns the pointer to the requested storage.

See Also

`kern_malloc`(D3X), **`kern_free`**(D3X), **`kmem_alloc`**(D3)

NAME

kern_free – free kernel memory space

SYNOPSIS

```
#include "sys/types.h"

kern_free(void *kern_v_addr);
```

DESCRIPTION

kern_free frees kernel virtual memory whose address is *kern_v_addr*. It frees the number of bytes that the **kern_malloc**(D3X) or **kern_calloc**(D3X) function assigned to this address.

Return Values

kern_calloc(D3X), **kmem_free**(D3), **kern_malloc**(D3X)

NAME

kern_malloc – allocate kernel virtual memory

SYNOPSIS

```
#include "sys/types.h"

void *kern_malloc(int num_bytes);
```

DESCRIPTION

kern_malloc allocates *num_bytes* of kernel virtual memory. If necessary, it sleeps until the entire requested memory is available. Therefore, do not call this function from an interrupt routine. Memory is not physically contiguous.

Return Values

Returns a pointer to the requested memory.

See Also

kern_calloc(D3X), **kern_free**(D3X), **kmem_alloc**(D3)

Note

Drivers that use DMA should use **kmem_alloc**(D3) to allocate buffers for DMA and free that memory with **kmem_free**(D3). For a discussion, see **kmem_alloc**(D3).

NAME

kmem_alloc – allocate space from kernel free memory

SYNOPSIS

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

void *kmem_alloc(size_t size, int flag);
```

Arguments

size	Number of bytes to allocate.
flag	Specifies whether the caller is willing to sleep waiting for memory, etc.

DESCRIPTION

kmem_alloc allocates *size* bytes of kernel memory and returns a pointer to the allocated memory. If *flag* is set to **KM_SLEEP**, the caller will sleep if necessary until the specified amount of memory is available. If *flag* is set to **KM_NOSLEEP**, the caller will not sleep, but **kmem_alloc** will return **NULL** if the specified amount of memory is not immediately available. **KM_PHYSCONTIG**: Allocate contiguous physical memory.

CAUTION: It is best to call **kmem_alloc** with this flag *only* at driver initialization time. Otherwise, it may sleep for a very long time.

KM_CACHEALIGN: Allocate the requested memory starting at a cache line boundary. This also pads the buffer out to a full cache line. Buffers that the driver will use for **DMA** must be cache-line aligned and padded to a full cache line.

Return Values

Upon successful completion, **kmem_alloc** returns a pointer to the allocated memory. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **kmem_alloc** returns a **NULL** pointer. If *size* is set to **0**, **kmem_alloc** returns **NULL** regardless of the value of *flag*.

USAGE

Kernel memory is a limited resource and should be used judiciously. Memory allocated using **kmem_alloc** should be freed as soon as possible. Drivers should not use local freelists for memory or similar schemes that cause the memory to be held for longer than necessary.

Since holding memory allocated using **kmem_alloc** for extended periods of time (e.g allocating memory at system startup and never freeing it) can have an adverse effect on overall memory usage and system performance, memory needed for such extended periods should be statically allocated whenever possible.

The address returned by a successful call to **kmem_alloc** is word-aligned.

Level

Base only if *flag* is set to **KM_SLEEP**.

Initialization, Base or Interrupt if *flag* is set to **`KM_NOSLEEP`**.

Synchronization Constraints

May sleep if *flag* is set to **`KM_SLEEP`**.

Driver-defined basic locks and read / write locks may be held across calls to this function if *flag* is **`KM_NOSLEEP`**, but may not be held if *flag* is **`KM_SLEEP`**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

Note

`kmem_alloc` and **`kmem_free`** are intended as replacements for **`kmem_malloc`** and **`kern_free`**. Drivers should use these routines rather than **`kern_malloc`** and **`kern_free`**.

REFERENCES

`kmem_free`(D3), **`kmem_zalloc`**(D3), Appendix A, Section A.2, "Data cache Write Back and Invalidation" of the *IRIX Device Driver Programming Guide*

NAME

kmem_free – free previously allocated kernel memory

SYNOPSIS

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

void kmem_free(void *addr, size_t size);
```

Arguments

addr	Address of the allocated memory to be returned.
size	Number of bytes to free.

DESCRIPTION

kmem_free returns *size* bytes of previously allocated kernel memory.

Return Values

None

USAGE

The *addr* argument must specify an address that was returned by a call to **kmem_alloc**(D3) or **kmem_zalloc**(D3).

The *size* argument must specify the same number of bytes as was allocated by the corresponding call to **kmem_alloc** or **kmem_zalloc**.

Together, the *addr* and *size* arguments must specify exactly one complete area of memory that was allocated by a call to **kmem_alloc** or **kmem_zalloc** (that is, the memory cannot be freed piecemeal).

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

kmem_alloc(D3), **kmem_zalloc**(D3)

NAME

kmem_zalloc – allocate and clear space from kernel free memory

SYNOPSIS

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

void *kmem_zalloc(size_t size, int flag);
```

Arguments

size	Number of bytes to allocate.
flag	Specifies whether the caller is willing to sleep waiting for memory, also other flags accepted by **kmem_alloc**.

DESCRIPTION

kmem_zalloc allocates *size* bytes of kernel memory, clears the memory by filling it with zeros, and returns a pointer to the allocated memory. If *flag* is set to **KM_SLEEP**, the caller will sleep if necessary until the specified amount of memory is available. If *flag* is set to **KM_NOSLEEP**, the caller will not sleep, but **kmem_zalloc** will return **NULL** if the specified amount of memory is not immediately available.

Return Values

Upon successful completion, **kmem_zalloc** returns a pointer to the allocated memory. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **kmem_zalloc** returns a **NULL** pointer. If *size* is set to **0**, **kmem_zalloc** returns **NULL** regardless of the value of *flag*.

USAGE

Kernel memory is a limited resource and should be used judiciously. Memory allocated using **kmem_zalloc** should be freed as soon as possible. Drivers should not use local freelists for memory or similar schemes that cause the memory to be held for longer than necessary.

Since holding memory allocated using **kmem_zalloc** for extended periods of time (e.g allocating memory at system startup and never freeing it) can have an adverse effect on overall memory usage and system performance, memory needed for such extended periods should be statically allocated whenever possible.

The address returned by a successful call to **kmem_zalloc** is word-aligned.

Level

Initialization or Base if *flag* is set to **KM_SLEEP**.

Initialization, Base or Interrupt if *flag* is set to **KM_NOSLEEP**.

Synchronization Constraints

May sleep if *flag* is set to **KM_SLEEP**.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function if *flag* is **KM_NOSLEEP**, but may not be held if *flag* is **KM_SLEEP**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

REFERENCES

`kmem_alloc`(D3), **`kmem_free`**(D3)

kvtophys(D3X)

NAME

kvtophys – get physical address of buffer data

SYNOPSIS

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

paddr_t kvtophys(void *kv);
```

Arguments

kv	Pointer to kernel virtual address.

DESCRIPTION

This function returns the physical address equivalent of the specified kernel virtual address. Mappings returned are only valid up to a page boundary.

Return Values

kvtophys returns *NULL* if *kv* is invalid; otherwise, a physical address is returned.

Caution

If *kv* is invalid, referencing the value returned by **kvtophys** could panic the system.

Level

Base or Interrupt.

Notes

Does not sleep.

Driver defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

See Also

getpagesize(2)

NAME

`linkb` – concatenate two message blocks

SYNOPSIS

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

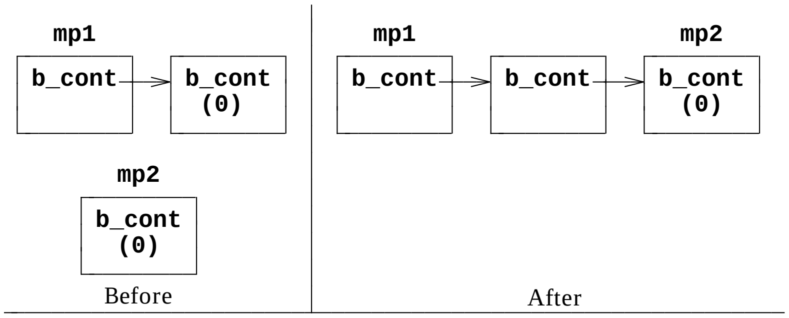
`void linkb(mblk_t *`*mp1***`, mblk_t *`***mp2***`);`**

Arguments

mp1	Pointer to the message to which *mp2* is to be added.
mp2	Pointer to the message to be added.

DESCRIPTION

`linkb` appends the message *mp2* to the tail of message *mp1*. The continuation pointer (**`b_cont`**) of the last message block in the first message is set to point to the second message:



`linkb(mp1, mp2);`

Return Values

None.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`msgb`(D4), **`unlinkb`**(D3)

NAME

LOCK – acquire a basic lock

SYNOPSIS

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

int LOCK(lock_t *lockp, pl_t pl);
```

Arguments

lockp	Pointer to the basic lock to be acquired.
pl	The interrupt priority level to be set while the lock is held by the caller.

DESCRIPTION

LOCK sets the interrupt priority level in accordance with the value specified by *pl* and acquires the lock specified by *lockp*. If the lock is not immediately available, the caller will wait until the lock is available. Some implementations may cause the caller to spin for the duration of the wait.

Return Values

Upon acquiring the lock, **LOCK** returns the previous mask for use by **UNLOCK**. Your driver should treat this return integer as an "opaque" and should not try to compare or do any operation.

USAGE

Because some implementations require that interrupts that might attempt to acquire the lock be blocked on the processor on which the lock is held, portable drivers must specify a *pl* value that is sufficient to block out any interrupt handler that might attempt to acquire this lock. See the description of the *min_pl* argument to **LOCK_ALLOC**(D3) for additional discussion and a list of the valid values for *pl*.

Level

Base or Interrupt.

Synchronization Constraints

Driver-defined sleep locks may be held across calls to this function.

Driver-defined basic locks and read / write locks may be held across calls to this function subject to the hierarchy.

Warnings

Basic locks are not recursive. A call to **LOCK** attempting to acquire a lock that is currently held by the calling context will result in deadlock.

Calls to **LOCK** should honor the ordering in order to avoid deadlock.

When called from interrupt level, the *pl* argument must not specify a priority level below the level at which the interrupt handler is running.

REFERENCES

LOCK_ALLOC(D3), **LOCK_DEALLOC**(D3), **TRYLOCK**(D3), **UNLOCK**(D3)

NAME

LOCK_ALLOC – allocate and initialize a basic lock

SYNOPSIS

```
#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>
```

```
lock_t *LOCK_ALLOC(uchar_t hierarchy, pl_t min_pl, lkinfo_t *lkinfop,
      int flag);
```

Arguments

hierarchy	Set to -1. Reserved for future use.
min_pl	Minimum priority level argument which asserts the minimum priority level that will be passed in with any attempt to acquire this lock [see **LOCK**(D3)].
lkinfop	Set to -1. Reserved for future use.
flag	Specifies whether the caller is willing to sleep waiting for memory.

DESCRIPTION

LOCK_ALLOC dynamically allocates and initializes an instance of a basic lock. The lock is initialized to the unlocked state.

If *flag* is set to **KM_SLEEP**, the caller will sleep if necessary until sufficient memory is available. If *flag* is set to **KM_NOSLEEP**, the caller will not sleep, but **LOCK_ALLOC** will return **NULL** if sufficient memory is not immediately available.

Return Values

Upon successful completion, **LOCK_ALLOC** returns a pointer to the newly allocated lock. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **LOCK_ALLOC** returns a **NULL** pointer.

min_pl Argument

The valid values for this argument are as follows:

plbase	Block no interrupts
pltimeout	Block functions scheduled by itimeout and dtimeout
pldisk	Block disk device interrupts
plstr	Block STREAMS interrupts
plhi	Block all interrupts

The notion of a *min_pl* assumes a defined order of priority levels. The following partial order is defined:

```
plbase < pltimeout <= pldisk,plstr <= plhi
```

The ordering of **pldisk** and **plstr** relative to each other is not defined.

Setting a given priority level will block interrupts associated with that level as well as any levels that are defined to be less than or equal to the specified level. In order to be portable a driver should not acquire locks at different priority levels where the relative order of those priority levels is not defined above.

The *min_pl* argument should specify a priority level that would be sufficient to block out any interrupt handler that might attempt to acquire this lock. In addition, potential deadlock problems involving multiple locks should be considered when defining the *min_pl* value. For example, if the normal order of acquisition of locks A and B (as defined by the lock hierarchy) is to acquire A first and then B, lock B should never be acquired at a priority level less than the *min_pl* for lock A. Therefore, the *min_pl* for lock B should be greater than or equal to the *min_pl* for lock A.

Note that the specification of *min_pl* with a **LOCK_ALLOC** call does not actually cause any interrupts to be blocked upon lock acquisition, it simply asserts that subsequent **LOCK** calls to acquire this lock will pass in a priority level at least as great as *min_pl*.

Level

Base only if *flag* is set to **KM_SLEEP**.

Initialization, Base or Interrupt if *flag* is set to **KM_NOSLEEP**.

Synchronization Constraints

May sleep if flag is set to **KM_SLEEP**.

Driver-defined basic locks and read / write locks may be held across calls to this function if *flag* is **KM_NOSLEEP** but may not be held if *flag* is **KM_SLEEP**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

REFERENCES

LOCK(D3), **LOCK_DEALLOC**(D3), **TRYLOCK**(D3), **UNLOCK**(D3)

LOCK_DEALLOC(D3)

NAME

`LOCK_DEALLOC` – deallocate an instance of a basic lock

SYNOPSIS

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

void LOCK_DEALLOC(lock_t *lockp);
```

Arguments

lockp	Pointer to the basic lock to be deallocated.

DESCRIPTION

`LOCK_DEALLOC` deallocates the basic lock specified by *lockp*.

Return Values

None.

USAGE

Attempting to deallocate a lock that is currently locked or is being waited for is an error and will result in undefined behavior.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks (other than the one being deallocated), read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`LOCK`(D3), **`LOCK_ALLOC`**(D3), **`TRYLOCK`**(D3), **`UNLOCK`**(D3)

NAME

makedevice – make device number from major and minor numbers

SYNOPSIS

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

dev_t makedevice(major_t majnum, minor_t minnum);
```

Arguments

majnum	Major number.
minnum	Minor number.

DESCRIPTION

The **makedevice** function creates a device number from major and minor device numbers.

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.

USAGE

makedevice should be used to create device numbers so that the driver will port easily to releases that treat device numbers differently.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Singlethreaded Example

In the following example, **makedevice** is used to create the device number selected during a clone open. If the **CLONEOPEN** flag is set (line 9), we search through the list of minor devices looking for one that is available (lines 10–11). If we find an unused minor, we break off the search, create a new device number, and store it in the memory location pointed to by **devp** (line 15). If no unused minor was found, we return the error **ENXIO**.

```
1  xxxopen(q, devp, oflag, sflag, crp)
2    queue_t *q;
3    dev_t *devp;
4    int oflag;
5    int sflag;
6    cred_t *crp;
7  {
8    minor_t minnum;

9    if (sflag == CLONEOPEN) {
10        for (minnum = 0; minnum < XXXMAXMIN; minnum++)
11            if (!INUSE(minnum))
12                break;
13        if (minnum >= XXXMAXMIN)
14            return(ENXIO);
15            SETINUSE(minnum);
16        *devp = makedevice(getemajor(*devp), minnum);
17    }
    ...
```

Multithreaded Example

In the following example, **makedevice** is used to create the device number selected during a clone open. If the **CLONEOPEN** flag is set (line 11), we lock the list of minor devices (line 12) and search through the list, looking for a minor device that is available (lines 13–14). If we find an unused minor, we break off the search, mark the minor as being in use (line 20), unlock the list, create a new device number, and store it in the memory location pointed to by **devp** (line 22). If no unused minor was found, we unlock the list and return the error **ENXIO**.

```
1  xxxopen(q, devp, oflag, sflag, crp)
2    queue_t *q;
3    dev_t *devp;
4    int oflag;
5    int sflag;
6    cred_t *crp;
7  {
8    minor_t minnum;
9    int pl;
10    extern lock_t *xxxminlock;
11    if (sflag == CLONEOPEN) {
12        pl = LOCK(xxxminlock, plstr);
13        for (minnum = 0; minnum < XXXMAXMIN; minnum++)
14            if (!INUSE(minnum))
15                break;
16        if (minnum >= XXXMAXMIN) {
17            UNLOCK(xxxminlock, pl);
18            return(ENXIO);
19        } else {
20            SETINUSE(minnum);
21            UNLOCK(xxxminlock, pl);
22            *devp = makedevice(getemajor(*devp), minnum);
23        }
24    }
        ...
```

REFERENCES

getemajor(D3), **geteminor**(D3), **getmajor**(D3), **getminor**(D3)

NAME

max – return the larger of two integers

SYNOPSIS

#include <sys/ddi.h>

int max(int *int1***, int** *int2***);**

Arguments

int1, *int2*	The integers to be compared.

DESCRIPTION

max compares two integers and returns the larger of two.

Return Values

The larger of the two integers.

USAGE

If the *int1* and *int2* arguments are not of the specified type the results are undefined.

This interface may be implemented in a way that causes the arguments to be evaluated multiple times, so callers should beware of side effects.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

min(D3)

NAME

min – return the lesser of two integers

SYNOPSIS

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

int min(int int1, int int2);
```

Arguments

int1, int2	The integers to be compared.

DESCRIPTION

min compares two integers and returns the lesser of the two.

Return Values

The lesser of the two integers.

USAGE

If the *int1* and *int2* arguments are not of the specified type the results are undefined.

This interface may be implemented in a way that causes the arguments to be evaluated multiple times, so callers should beware of side effects.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

max(D3)

NAME

msgdsize – return number of bytes of data in a message

SYNOPSIS

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

int msgdsize(mblk_t *mp);
```

Arguments

mp	Pointer to the message to be evaluated.

DESCRIPTION

msgdsize counts the number of bytes of data in the message pointed to by *mp*. Only bytes included in message blocks of type **M_DATA** are included in the count.

Return Values

The number of bytes of data in the message.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **insq**(D3) for an example of **msgdsize**.

REFERENCES

msgb(D4)

NAME

msgpullup – concatenate bytes in a message

SYNOPSIS

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

mblk_t *msgpullup(mblk_t *mp, int len);
```

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. All message blocks that remain in the original message once *len* bytes have been concatenated and aligned (including any partial message blocks) are copied and linked to the end of the new message, so that the length of the new message is equal to the length of the original 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, **msgpullup** returns a pointer to the new message. On failure, **msgpullup** returns **NULL**.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

allocb(D3), **msgb**(D4)

NAME

ngeteblk – get an empty buffer of the specified size

SYNOPSIS

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

buf_t *ngeteblk(size_t bsize);
```

Arguments

bsize	Size of the buffer being requested.

DESCRIPTION

ngeteblk retrieves a buffer [see **buf**(D4)] of size *bsize* from the buffer cache and returns a pointer to the buffer header. If a buffer is not available, **ngeteblk** dynamically allocates one. If memory is not immediately available, **ngeteblk** will sleep until enough memory has been freed to allocate the buffer.

Return Values

A pointer to the buffer header structure is returned.

USAGE

When the driver **strategy**(D2) routine receives a buffer header from the kernel, all the necessary members are already initialized. However, when a driver allocates buffers for its own use, it must set up some of the members before calling its **strategy** routine.

The following list describes the state of these members when the buffer header is received from **ngeteblk**:

b_flags	is set to indicate the transfer is from the user's buffer to the kernel. The driver must set the **B_READ** flag if the transfer is from the kernel to the user's buffer.
b_edev	is set to **NODEV** and must be initialized by the driver.
b_bcount	is set to *bsize*.
b_un.b_addr	is set to the buffer's virtual address.
b_blkno	is not initialized by **ngeteblk**, and must be initialized by the driver

Typically, block drivers do not allocate buffers. The buffer is allocated by the kernel, and the associated buffer header is used as an argument to the driver **strategy** routine. However, to implement some special features, such as **ioctl**(D2) commands that perform I/O, the driver may need its own buffer space. The driver can get the buffer space from the system by using **geteblk**(D3) or **ngeteblk**. Or the driver can choose to use its own memory for the buffer and only allocate a buffer header with **getrbuf**(D3).

Note that buffers allocated via **ngeteblk** must be freed using either **brelse**(D3) or **biodone**(D3).

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

REFERENCES

biodone(D3), **brelse**(D3), **buf**(D4), **geteblk**(D3)

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.

DESCRIPTION

The **noenable** function prevents the service routine of the queue pointed to by *q* from being scheduled for service by **insq**(D3), **putbq**(D3), or **putq**(D3), when enqueuing a message that is not a high priority message.

Return Values

None

USAGE

The high-priority-only message restriction can be lifted with the **enableok**(D3) function.

noenable does not prevent the queue's service routine from being scheduled when a high priority message is enqueued, or by an explicit call to **qenable**(D3).

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

enableok(D3), **insq**(D3), **putbq**(D3), **putq**(D3), **qenable**(D3), **queue**(D4), **srv**(D2)

NAME

OTHERQ – get a 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.

DESCRIPTION

The **OTHERQ** function returns a pointer to the other of the two **queue** structures that make up an instance of a STREAMS module or driver.

Return Values

OTHERQ returns a pointer to a queue's partner.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Multithreaded drivers may hold driver-defined basic locks, read / write locks, and sleep locks across calls to this function.

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(queue_t *q, long min, long max, ulong_t hi, ulong_t lo)
{
   pl_t pl;     /* for multi-threaded drivers */
   (void) strqset(q, QMINPSZ, 0, min);
   (void) strqset(q, QMAXPSZ, 0, max);
   (void) strqset(q, QHIWAT, 0, hi);
   (void) strqset(q, QLOWAT, 0, lo);
   (void) strqset(OTHERQ(q), QMINPSZ, 0, min);
   (void) strqset(OTHERQ(q), QMAXPSZ, 0, max);
   (void) strqset(OTHERQ(q), QHIWAT, 0, hi);
   (void) strqset(OTHERQ(q), QLOWAT, 0, lo);
}
```

REFERENCES

RD(D3), **WR**(D3)

NAME

pcmsg – test whether a message is a priority control message

SYNOPSIS

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

int pcmsg(uchar_t type);
```

Arguments

type	The type of message to be tested.

DESCRIPTION

The **pcmsg** function tests the type of message to determine if it is a priority control message (also known as a high priority message).

Return Values

pcmsg returns 1 if the message is a priority control message and 0 if the message is any other type.

USAGE

The **db_type** field of the **datab**(D4) structure contains the message type. This field may be accessed through the message block using **mp->b_datap->db_type**.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

The service routine processes messages on the queue. If the message is a high priority message, or if it is a normal message and the stream is not flow-controlled, the message is processed and passed along in the stream. Otherwise, the message is placed back on the head of the queue and the service routine returns.

```
xxxsrv(q)
    queue_t *q;
{
    mblk_t *mp;

    while ((mp = getq(q)) != NULL) {
        if (pcmsg(mp->b_datap->db_type) || canputnext(q->g_next)) {
            /* process message */
            putnext(q, mp);
        } else {
            putbq(q, mp);
            return;
        }
    }
14    }
```

REFERENCES

allocb(D3), **datab**(D4), **msgb**(D4), **messages**(D5)

NAME

phalloc – allocate and initialize a pollhead structure

SYNOPSIS

```
#include <sys/poll.h>
#include <sys/kmem.h>
#include <sys/ddi.h>

struct pollhead *phalloc(int flag);
```

Arguments

flag	Specifies whether the caller is willing to sleep waiting for memory.

DESCRIPTION

phalloc allocates and initializes a **pollhead** structure for use by non-STREAMS character drivers that wish to support polling. If *flag* is set to **KM_SLEEP**, the caller will sleep if necessary until sufficient memory is available. If *flag* is set to **KM_NOSLEEP**, the caller will not sleep, but **phalloc** will return **NULL** if sufficient memory is not immediately available.

Return Values

On success, **phalloc** returns a pointer to the newly allocated **pollhead** structure. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **phalloc** returns a **NULL** pointer.

USAGE

On systems where the **phalloc** function is available, DDI / DKI conforming drivers should only use **pollhead** structures which have been allocated and initialized using **phalloc**. Use of **pollhead** structures which have been obtained by any other means is prohibited on such systems.

Level

Base only if *flag* is set to **KM_SLEEP**.

Initialization, Base or Interrupt if *flag* is set to **KM_NOSLEEP**.

Synchronization Constraints

May sleep if flag is set to **KM_SLEEP**.

Driver-defined basic locks and read / write locks may be held across calls to this function if *flag* is **KM_NOSLEEP** but may not be held if *flag* is **KM_SLEEP**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

REFERENCES

chpoll(D2), **phfree**(D3)

NAME

phfree – free a pollhead structure

SYNOPSIS

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

void phfree(struct pollhead *php);
```

Arguments

php	Pointer to the **pollhead** structure to be freed.

DESCRIPTION

phfree frees the **pollhead** structure specified by *php*.

Return Values

None.

USAGE

The structure pointed to by *php* must have been previously allocated by a call to **phalloc**(D3).

On systems where the **phalloc** function is available, DDI / DKI conforming drivers should only use **pollhead** structures which have been allocated and initialized using **phalloc**. Use of **pollhead** structures which have been obtained by any other means is prohibited on such systems.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

chpoll(D2), **phalloc**(D3)

NAME

physiock – validate and issue a raw I/O request

SYNOPSIS

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

int physiock(void (*strat)(), buf_t *bp, dev_t dev, int rwflag,
      daddr_t nblocks, uio_t *uiop);
```

Arguments

strat	Address of the driver **strategy**(D2) routine, or similar function.
bp	Pointer to the **buf**(D4) structure describing the I/O request.
dev	External device number.
rwflag	Flag indicating whether the access is a read or a write.
nblocks	Number of blocks that the logical device *dev* can support.
uiop	Pointer to the **uio**(D4) structure that defines the user space of the I/O request.

DESCRIPTION

physiock is called by the character interface **ioctl**(D2), **read**(D2), and **write**(D2) routines of block drivers to help perform unbuffered I/O while maintaining the buffer header as the interface structure.

Return Values

physiock returns 0 if the result is successful, or the appropriate error number on failure. If a partial transfer occurs, the **uio** structure is updated to indicate the amount not transferred and an error is returned. **physiock** returns the **ZNOPC** error if an attempt is made to read beyond the end of the device. If a read is performed at the end of the device, 0 is returned. **ZNOSPC** is also returned if an attempt is made to write at or beyond the end of a the device. **EFAULT** is returned if user memory is not valid. **EAGAIN** is returned if **physiock** could not lock all of the pages.

USAGE

physiock performs the following functions:

verifies the requested transfer is valid by checking if the offset is at or past the end of the device (this check is bypassed if the size parameter argument **nblocks** is zero) and that the offset is a multiple of 512

sets up a buffer header describing the transfer

faults pages in and locks the pages impacted by the I/O transfer so they can't be swapped out

calls the driver **strategy** routine passed to it (*strat*)

sleeps until the transfer is complete and is awakened by a call to **biodone**(D3) from the driver's I/O completion handler performs the necessary cleanup and updates, then returns to the driver routine

A transfer using **`physiock`** is considered valid if the specified data location exists on the device, and the user has specified a storage area large enough that exists in user memory space.

If *bp* is set to **`NULL`**, a buffer is allocated temporarily and freed after the transfer completes.

If *rwflag* is set to **`B_READ`**, the direction of the data transfer will be from the kernel to the user's buffer. If *rwflag* is set to **`B_WRITE`**, the direction of the data transfer will be from the user's buffer to the kernel.

One block is equal to **`NBPSCTR`** bytes. **`NBPSCTR`** is defined in **`sys/param.h`**.

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function.

REFERENCES

`buf`(D4), **`ioctl`**(D2), **`read`**(D2), **`strategy`**(D2), **`uio`**(D4), **`write`**(D2), **`uiophysio`**(D3X)

NAME

pio_andb_rmw – byte VME-bus read-modify-write cycle routines

SYNOPSIS

pio_andb_rmw(piomap_t ****piomap*, iopaddr_t** *iopaddr***, unsigned char** *mask***);**

DESCRIPTION

pio_andb_rmw performs an atomic VME-bus read-modify-write operation. This function reads a byte from the address given by the *iopaddr* argument. The function then ANDs the byte, with the mask specified by *mask*, and writes the result to the address, *iopaddr*. To prevent any other VME-bus cycles during this operation, this function locks the VME bus. **piomap** is the PIO map returned from **pio_mapalloc**(D3X).

Note

The address must be correctly aligned for the given transfer.

See Also

pio_orb_rmw(D3X), **pio_andh_rmw**(D3X), **pio_andw_rmw**(D3X)

NAME

pio_andh_rmw – half-word VME-bus read-modify-write cycle routine

SYNOPSIS

pio_andh_rmw(piomap_t ***_piomap_, iopaddr_t** _pioaddr_**, unsigned short** _mask_**);**

DESCRIPTION

pio_andh_rmw performs an atomic VME-bus read-modify-write operation. This function reads a half-word from the address given by the *pioaddr* argument. The function then ANDs the half-word with the mask specified by *mask*, and writes the result to the address, *pioaddr*. To prevent any other VME-bus cycles during this operation, this function locks the VME bus. **piomap** is the PIO map returned from **pio_mapalloc**(D3X).

Note

The address must be correctly aligned for the given transfer.

See Also

pio_orb_rmw(D3X), **pio_andh_rmw**(D3X), **pio_andw_rmw**(D3X)

pio_andw_rmw(D3X)

NAME

pio_andw_rmw – word VME-bus read-modify-write cycle routines

SYNOPSIS

pio_andw_rmw(piomap_t ***_piomap_, iopaddr_t** _pioaddr_**, unsigned long** _mask_**);**

DESCRIPTION

pio_andw_rmw perform an atomic VME-bus read-modify-write operation. This function reads a word from the address given by the _pioaddr_ argument. The function then ANDs the word with the mask specified by _mask_, and writes the result to the address, _pioaddr_. To prevent any other VME-bus cycles during this operation, this function locks the VME bus. **piomap** is the PIO map returned from **pio_mapalloc**(D3X).

Note

The address must be correctly aligned for the given transfer.

See Also

pio_orb_rmw(D3X), **pio_andb_rmw**(D3X), **pio_andh_rmw**(D3X)

NAME

pio_badaddr – check for bus error when reading an address

SYNOPSIS

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

int pio_badaddr (piomap_t* piomap, iopaddr_t iopaddr, int len);
```

Arguments

piomap	The PIO map returned from **pio_mapalloc**(D3X).
iopaddr	The VME bus address to be probed.
len	The size in bytes to probe the VME bus address.

DESCRIPTION

Call **pio_badaddr** to determine whether you can read specified address location. Typically, you call **pio_badaddr** from a VME device's **edtinit()** function to determine whether a device is still on the present system.

Return Values

Returns a zero if the probe was successful.

See Also

pio_mapalloc(D3X), **pio_mapfree**(D3X), **pio_mapaddr**(D3X), **pio_wbadaddr**(D3X), **pio_bcopyin** (Da3x), **pio_bcopyout**(D3X)

NAME

pio_bcopyin – copy data from VME bus address to kernel's virtual space

SYNOPSIS

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

int pio_bcopyin(piomap_t *piomap, iopaddr_t iopaddr, void *kvaddr,
                        int size, int itmsz, int flag);
```

Arguments

piomap	The PIO map returned from **pio_mapalloc**(D3X).
iopaddr	The VME bus address.
kvaddr	The kernel's virtual address.
size	The byte count of the transfer.
itmsz	The element size of each read or write of the VME bus.
flag	*PIO_NOSLEEP* if this call shouldn't block.

DESCRIPTION

pio_bcopyin copies data from the VME bus address space to the kernel virtual address space.

Return Values

The number of bytes transferred.

See Also

pio_mapalloc(D3X), **pio_mapfree**(D3X), **pio_mapaddr**(D3X), **pio_badaddr**(D3X), **pio_wbadaddr**(D3X), **pio_bcopyout**(D3X)

NAME

pio_bcopyout – copy data from kernel's virtual space to VME bus address

SYNOPSIS

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

int pio_bcopyout (piomap_t *piomap, iopaddr_t iopaddr, void *kvaddr,
                  int size, int itmsz, int flag);
```

Arguments

piomap	The PIO map returned from **pio_mapalloc**(D3X).
iopaddr	The VME bus address.
kvaddr	The kernel's virtual address.
size	The byte count of the transfer.
itmsz	The element size of each read or write of the VME bus.
flag	*PIO_NOSLEEP* if this call shouldn't block.

DESCRIPTION

pio_bcopyin copies data from the kernel virtual address space to the VME bus address space.

Return Values

The number of bytes transferred.

See Also

pio_mapalloc(D3X), **pio_mapfree**(D3X), **pio_mapaddr**(D3X), **pio_badaddr**(D3X), **pio_wbadaddr**(D3X), **pio_bcopyin**(D3X)

pio_mapaddr(D3X)

NAME

pio_mapaddr – used with FIXED maps to generate a kernel pointer to VME bus space

SYNOPSIS

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

caddr_t pio_mapaddr (piomap_t *piomap, iopaddr_t addr);
```

Arguments

piomap	The PIO map returned from **pio_mapalloc**(D3X).
addr	The BME address to be mapped.

DESCRIPTION

pio_mapaddr converts a VME address to a virtual address.

Return Values

A pointer which when accessed, will access the VME bus location specified by addr.

See Also

pio_mapfree(D3X), **pio_mapalloc**(D3X), **pio_badaddr**(D3X), **pio_wbadaddr**(D3X), **pio_bcopyin**(D3X), **pio_bcopyout**(D3X)

NAME

pio_mapalloc – allocate a PIO map

SYNOPSIS

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

piomap_t *
pio_mapalloc(uint bus, uint adap, iospace_t *iospace, int flag, char *name);
```

Arguments

bus	The type of bus the map is for, ADAP_VME from *edt.h*.
adap	Identifies which VME bus. The Challenge series supports up to five.
iospace	This defines the space on the VME bus to mapped, a16s, a24n, etc.
flag	*PIOMAP_FIXED* or *PIOMAP_UNFIXED*.
name	A character string used to identify the map. Useful for debugging a driver.

DESCRIPTION

pio_mapalloc creates PIO maps used to access VME bus space from a driver. PIO maps can be *FIXED* or *UNFIXED*. *FIXED* maps provide the driver with a kernel address which can be used as a normal pointer to access VME bus space. *UNFIXED* maps require the use of special routines access to VME bus space.

Return Values

A pointer to a *piomap_t* type structure which is used with the reset of the routines.

See Also

pio_mapfree(D3X), **pio_mapaddr**(D3X), **pio_badaddr**(D3X), **pio_wbadaddr**(D3X), **pio_bcopyin**(D3X), **pio_bcopyout**(D3X)

NAME

pio_mapfree – free up a previously allocated PIO map

SYNOPSIS

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

void pio_mapfree(piomap_t *piomap);
```

Arguments

piomap The PIO map to be freed.

DESCRIPTION

Pio_mapfree frees the specified PIO map which is previously allocated by **pio_mapalloc**(D3X).

Return Values

None

See Also

pio_mapalloc(D3X), **pio_mapaddr**(D3X), **pio_badaddr**(D3X), **pio_wbadaddr**(D3X), **pio_bcopyin**(D3X), **pio_bcopyout**(D3X)

NAME

pio_orb_rmw – VME-bus read-modify-write cycle routines

SYNOPSIS

pio_orb_rmw(piomap_t ***piomap*, iopaddr_t** *pioaddr*, **unsigned char** *mask***);**

DESCRIPTION

This function perform VME-bus atomic read-modify-write operations. **pio_orb_rmw**(D3X), **pio_orh_rmw**(D3X), or **pio_orw_rmw**(D3X) read a byte, half-word, or word (respectively) from the address pointed to by *pioaddr*. The routine then ORs the byte, half-word, or word with the mask in *mask* and writes the result to the address, *pioaddr* (overwriting the original value). **piomap** is the PIO map returned from **pio_mapalloc**(D3X).

The address, *pioaddr*, must be correctly aligned for the given transfer.

Return Values

None

Note

To prevent any other VME-bus cycles during this operation, these routines lock the VME bus.

pio_orh_rmw(D3X)

NAME

pio_orh_rmw – VME-bus read-modify-write cycle routines

SYNOPSIS

pio_orh_rmw(piomap_t **piomap*, **iopaddr_t** *pioaddr*, **unsigned short** *mask***);**

DESCRIPTION

This function perform VME-bus atomic read-modify-write operations. **pio_orb_rmw**(D3X), **pio_orh_rmw**(D3X), or **pio_orw_rmw**(D3X) read a byte, half-word, or word (respectively) from the address pointed to by *pioaddr*. The routine then ORs the byte, half-word, or word with the mask in *mask* and writes the result to the address, *pioaddr* (overwriting the original value). **piomap** is the PIO map returned from **pio_mapalloc**(D3X).

The address, *pioaddr*, must be correctly aligned for the given transfer.

Return Values

None

Note

To prevent any other VME-bus cycles during this operation, these routines lock the VME bus.

NAME

`pio_orw_rmw` – VME-bus read-modify-write cycle routines

SYNOPSIS

`pio_orw_rmw (piomap_t *`*piomap***`, iopaddr_t`** *pioaddr***`, unsigned long`** *mask***`);`**

DESCRIPTION

This function perform VME-bus atomic read-modify-write operations. **`pio_orb_rmw`**(D3X), **`pio_orh_rmw`**(D3X), or **`pio_orw_rmw`**(D3X) read a byte, half-word, or word (respectively) from the address pointed to by *pioaddr*. The routine then ORs the byte, half-word, or word with the mask in *mask* and writes the result to the address, *pioaddr* (overwriting the original value). **`piomap`** is the PIO map returned from **`pio_mapalloc`**(D3X).

The address, *pioaddr*, must be correctly aligned for the given transfer.

Return Values

None

Note

To prevent any other VME-bus cycles during this operation, these routines lock the VME bus.

NAME

pio_wbadaddr – check for bus error when writing to an address

SYNOPSIS

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

int pio_wbadaddr (piomap_t *piomap, iopaddr_t iopaddr, int len);
```

Arguments

piomap	The PIO map returned from **pio_mapalloc**(D3X).
iopaddr	The VME bus address to be probed.
len	The size in bytes to probe the VME bus address.

DESCRIPTION

Call **pio_badaddr** to determine whether you can write to specified address location. Typically, you call **pio_wbadaddr** from a VME device's **edtinit()** function to determine whether a device is still on the present system.

Return Values

Returns a zero if the probe was successful.

See Also

pio_mapalloc(D3X), **pio_mapfree**(D3X), **pio_badaddr**(D3X), **pio_badaddr**(D3X), **pio_bcopyin**(D3X), **pio_bcopyout**(D3X)

NAME

`pollwakeup` – inform polling processes that an event has occurred

SYNOPSIS

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

void pollwakeup(struct pollhead *php, short event);
```

Arguments

php	Pointer to a **`pollhead`** structure.
event	Event to notify the process about.

DESCRIPTION

The **`pollwakeup`** function provides non-STREAMS character drivers with a way to notify processes polling for the occurrence of an event.

Return Values

None

USAGE

`pollwakeup` should be called from the driver for each occurrence of an event. Events are described in **`chpoll`**(D2).

The **`pollhead`** structure will usually be associated with the driver's private data structure for the particular minor device where the event has occurred.

`pollwakeup` should only be called with one event at a time.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`chpoll`(D2), **`poll`**(2)

pptophys(D3X)

NAME

pptophys – convert page pointer to physical address

SYNOPSIS

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

paddr_t pptophys(struct pfdat *pp);
```

Arguments

pp	Pointer to the page structure

DESCRIPTION

pptophys converts a pointer to a page structure to a physical address.

Return Values

The physical address represented by the page (pfdat) structure referenced by **pp**.

Block drivers can use this address for physical DMA operations during paged-I/O requests (see **getnextpg**(D3X)).

Level

Base or Interrupt.

Notes

Does not sleep.

See Also

strategy(D2), **getnextpg**(D3X), **buf**(D4)

NAME

`proc_ref` – obtain a reference to a process for signaling

SYNOPSIS

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

void *proc_ref(void);
```

DESCRIPTION

A non-STREAMS character driver can call **`proc_ref`** to obtain a reference to the process in whose context it is running.

Return Values

`proc_ref` returns an identifier that can be used in calls to **`proc_signal`** and **`proc_unref`**(D3).

USAGE

The value returned can be used in subsequent calls to **`proc_signal`**(D3) to post a signal to the process. The return value should not be used in any other way (that is, the driver should not attempt to interpret its meaning).

Processes can exit even though they are referenced by drivers. In this event, reuse of the identifier will be deferred until all driver references are given up.

There must be a matching call to **`proc_unref`** for every call to **`proc_ref`**, when the driver no longer needs to reference the process. This is typically done as part of **`close`**(D2) processing.

This function requires user context.

Level

Base only.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`proc_signal`(D3), **`proc_unref`**(D3)

NAME

`proc_signal` – send a signal to a process

SYNOPSIS

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

int proc_signal(void *pref, int sig);
```

Arguments

pref	Identifier obtained by a previous call to **`proc_ref`**(D3).
sig	Signal number to be sent.

DESCRIPTION

The **`proc_signal`** function can be used to post a signal to the process represented by *pref*. This will interrupt any process blocked in **`SV_WAIT_SIG`**(D3) or **`SLEEP_LOCK_SIG`**(D3) at the time the signal is posted, causing those functions to return prematurely in most cases. If the process has exited then this function has no effect.

Return Values

If the process still exists, 0 is returned. Otherwise, –1 is returned to indicate that the process no longer exists.

USAGE

Valid signal numbers are listed in **`signals`**(D5).

STREAMS drivers and modules should not use this mechanism for signaling processes. Instead, they can send **`M_SIG`** or **`M_PCSIG`** STREAMS messages to the stream head.

`proc_signal` must not be used to send **`SIGTSTP`** to a process.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`proc_ref`(D3), **`proc_unref`**(D3), **`signals`**(D5)

NAME

`proc_unref` – release a reference to a process

SYNOPSIS

`#include <sys/ddi.h>`

`void proc_unref(void *`*pref***`);`**

Arguments

pref	Identifier obtained by a previous call to **`proc_ref`**(D3).

DESCRIPTION

The **`proc_unref`** function can be used to release a reference to a process identified by the parameter *pref*.

Return Values

None

USAGE

There must be a matching call to **`proc_unref`** for every previous call to **`proc_ref`**(D3).

Processes can exit even though they are referenced by drivers. In this event, reuse of *pref* will be deferred until all driver references are given up.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`proc_ref`(D3), **`proc_signal`**(D3)

NAME

psema – perform a "P" or wait semaphore operation

SYNOPSIS

```
#include "sys/types.h"
#include "sys/param.h"
#include "sys/sema.h"

psema(sema_t *semap, int priority);
```

DESCRIPTION

psema performs a "P" semaphore operation on the given semaphore. The value associated with the semaphore is decremented by 1. If the semaphore value then becomes less than 0, the process goes to sleep and gives up the CPU.

Use *semap* to pass **psema** a pointer to the semaphore you want to decrement. Use *priority* to specify the priority you want to assign to the sleeping process when it is awakened. The *priority* argument also determines whether signals can awaken the process. If the *priority* value is greater than *PZERO*, it is breakable; otherwise it is not. If the process is awakened by a signal, then the semaphore value is incremented and is allowed to continue. If *PCATCH* is ORed into the *priority*, **psema** returns –1, and the process continues after the call; otherwise, control returns to the last point in the kernel where the process signal context was saved, usually at the beginning of the system call.

To initialize and allocate a semaphore, call **initnsema** or **initnsema_mutex**.

Return Values

psema returns –1 if a breakable sleep is interrupted by a signal and *PCATCH* is set. Otherwise **psema** returns 0.

Note

psema may cause the calling process to sleep; it must not be called from within an interrupt procedure.

See Also

sleep(D3), **SLEEP_LOCK**(D3)

NAME

ptob – convert size in pages to size in bytes

SYNOPSIS

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

ulong_t ptob(ulong_t numpages);
```

Arguments

numpages	Size in pages to convert to equivalent size in bytes.

DESCRIPTION

ptob returns the number of bytes that are contained in the specified number of pages.

Return Values

The return value is the number of bytes in the specified number of pages.

USAGE

There is no checking done on the input value and overflow is not detected.

In the case of a page count whose corresponding byte count cannot be represented by a **ulong_t** the higher order bits are truncated.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

If the page size is 2048, then **ptob**(2) returns 4096. **ptob(0)** returns 0.

REFERENCES

btop(D3), **btopr**(D3)

NAME

putbq – place a message at the head of a queue

SYNOPSIS

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

int putbq(queue_t *q, mblk_t *bp);
```

Arguments

q	Pointer to the queue.
bp	Pointer to the message.

DESCRIPTION

putbq puts a message back at the head of a queue. If messages of a higher priority are on the queue, then *bp* is placed at the head of its corresponding priority band. See **srv**(D2) for more information about message priorities.

All flow control parameters are updated. The queue's service routine is scheduled if it has not been disabled by a previous call to **noenable**(D3).

Return Values

putbq returns 1 on success and 0 on failure.

USAGE

putbq is usually called when **bcanput**(D3) or **canput**(D3) determines that the message cannot be passed on to the next stream component.

putbq can fail if there is not enough memory to allocate the accounting data structures used with messages whose priority bands are greater than zero.

High priority messages should never be put back on a queue from within a service routine.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **bufcall**(D3) for an example of **putbq**.

REFERENCES

bcanput(D3), **canput**(D3), **getq**(D3), **insq**(D3), **msgb**(D4), **putq**(D3), **queue**(D4), **rmvq**(D3), **srv**(D2)

NAME

`putctl` – send a control message to a queue

SYNOPSIS

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

int putctl(queue_t *q, int type);
```

Arguments

q	Pointer to the queue to which the message is to be sent.
type	Message type (must be a control type).

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`**(D2) routine of the queue pointed to by *q*, passing it the allocated message.

Return Values

On success, 1 is returned. Otherwise, if *type* is a data type, or if a message block cannot be allocated, 0 is returned.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function.

Examples

The **`send_ctl`** routine is used to pass control messages downstream. **`M_BREAK`** messages are handled with **`putctl`** (line 9). **`putctl1`** (line 11) is used for **`M_DELAY`** messages, so that *param* can be used to specify the length of the delay. If an invalid message type is detected, **`send_ctl`** returns 0, indicating failure (line 13).

```
 1  int
 2  send_ctl(wrq, type, param)
 3     queue_t *wrq;
 4     uchar_t type;
 5     uchar_t param;
 6  {
 7     switch (type) {
 8     case M_BREAK:
 9             return(putctl(wrq->q_next, M_BREAK));
10     case M_DELAY:
11             return(putctl1(wrq->q_next, M_DELAY, param));
12     default:
```

```
         return(0);
   }
}
```

REFERENCES

put(D2), **putctl1**(D3)

NAME

putctl1 – send a control message with a one-byte parameter to a queue

SYNOPSIS

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

int putctl1(queue_t *q, int type, int param);
```

Arguments

q	Pointer to the queue to which the message is to be sent.
type	Message type (must be a control type).
param	One-byte parameter.

DESCRIPTION

putctl1, like **putctl**(D3), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *param* parameter can be used, for example, to specify the signal number when an **M_PCSIG** message is being sent. **putctl1** fails if **type** is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putctl1** calls the **put**(D2) routine of the queue pointed to by *q*, passing it the allocated message.

Return Values

On success, 1 is returned. Otherwise, if *type* is a data type, or if a message block cannot be allocated, 0 is returned.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function.

Examples

See **putctl**(D3) for an example of **putctl1**.

REFERENCES

put(D2), **putctl**(D3)

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	Pointer to the message to be passed.

DESCRIPTION

The **putnext** function is used to pass a message to the **put**(D2) routine of the next queue (*q*–>*q_next*) in the stream.

Return Values

Ignored

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function.

Examples

See **allocb**(D3) for an example of **putnext**.

REFERENCES

put(D2)

NAME

putq – put a message on a queue

SYNOPSIS

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

int putq(queue_t *q, mblk_t *bp);
```

Arguments

q	Pointer to the queue.
bp	Pointer to the message.

DESCRIPTION

putq is used to put messages on a queue after the **put**(D2) routine has finished processing the message. The message is placed after any other messages of the same priority, and flow control parameters are updated. The queue's service routine is scheduled if it has not been disabled by a previous call to **noenable**(D3), or if the message being enqueued has greater than normal priority (that is, it is not in band zero).

Return Values

putq returns 1 on success and 0 on failure.

USAGE

putq can fail if there is not enough memory to allocate the accounting data structures used with messages whose priority bands are greater than zero.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **datamsg**(D3) for an example of **putq**.

REFERENCES

getq(D3), **insq**(D3), **msgb**(D4), **put**(D2), **putbq**(D3), **queue**(D4), **rmvq**(D3), **srv**(D2)

qenable(D3)

NAME

qenable – schedule a queue's service routine to be run

SYNOPSIS

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

void qenable(queue_t *q);
```

Arguments

q	Pointer to the queue.

DESCRIPTION

qenable puts the queue pointed to by *q* on the linked list of those whose service routines are ready to be called by the STREAMS scheduler.

Return Values

None

USAGE

qenable works regardless of whether the service routine has been disabled by a prior call to **noenable**(D3).

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **enableok**(D3) for an example of **qenable**.

REFERENCES

enableok(D3), **noenable**(D3), **queue**(D4), **srv**(D2)

NAME

qreply – send a message in the opposite direction in a stream

SYNOPSIS

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

void qreply(queue_t *q, mblk_t *bp);
```

Arguments

q	Pointer to the queue from which the message is being sent.
bp	Pointer to the message to be sent in the opposite direction.

DESCRIPTION

qreply sends a message in the opposite direction from that which *q* is pointing. It calls the **OTHERQ**(D3) function to find *q*'s partner, and passes the message by calling the **put**(D2) routine of the next queue in the stream after *q*'s partner.

Return Values

None

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function.

Examples

See **put**(D2) for an example of **qreply**.

REFERENCES

OTHERQ(D3), **put**(D2), **putnext**(D3)

qsize(D3)

NAME

qsize – find the number of messages on a queue

SYNOPSIS

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

int qsize(queue_t *q);
```

Arguments

q	Pointer to the queue to be evaluated.

DESCRIPTION

qsize evaluates the queue pointed to by *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 number of messages on the queue.

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

msgb(D4), **queue**(D4)

NAME

RD – get a 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 queue whose read queue is to be returned.

DESCRIPTION

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

Return Values

The pointer to the read queue.

USAGE

Note that when **RD** is passed a read queue pointer as an argument, it returns a pointer to this read queue.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See the **put**(D2) function page for an example of **RD**.

REFERENCES

OTHERQ(D3), **WR**(D3)

rmalloc(D3)

NAME

rmalloc – allocate space from a private space management map

SYNOPSIS

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

ulong_t rmalloc(struct map *mp, size_t size);
```

Arguments

mp	Pointer to the map from which space is to be allocated.
size	Number of units of space to allocate.

DESCRIPTION

rmalloc allocates space from the private space management map pointed to by *mp*.

Return Values

Upon successful completion, **rmalloc** returns the base of the allocated space. If *size* units cannot be allocated, 0 is returned.

USAGE

Drivers can use **rmalloc** to allocate space from a previously allocated and initialized private space management map.

On systems where the **rmallocmap** function is available, the map must have been allocated by a call to **rmallocmap**(D3) and the space managed by the map must have been added using **rmfree**(D3) prior to the first call to **rmalloc** for the map.

On systems where the **rmallocmap** function is not available, the map must be initially allocated either as a data array, or by the **kmem_alloc**(D3) function. The map must have been initialized by a call to **rminit**(D3) and the space managed by the map must have been added using **rmfree**(D3) prior to the first call to **rmalloc** for the map.

size specifies the amount of space to allocate and is in arbitrary units. The driver using the map places whatever semantics on the units are appropriate for the type of space being managed. For example, units may be byte addresses, pages of memory, or blocks on a device.

The system allocates space from the memory map on a first-fit basis and coalesces adjacent space fragments when space is returned to the map by **rmfree**.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

The following example is a simple memory map, but it illustrates the principles of map management. A driver declares a map table (line 4) and initializes the map table by calling both the **rminit** and **rmfree** functions. There are 35 entries in the map table, 32 of which can be used to represent space allocated. In the driver's **start**(D2) routine, we allocate 16 Kbytes of memory using **kmem_alloc**(D3) (line 8). This is the space to be managed. Then we call **rminit** to establish the number of slots or entries in the map (line 10), and **rmfree** to populate the map with the space it is to manage (line 11).

In the driver's **read**(D2) and **write**(D2) routines, we use **rmalloc** to allocate buffers for data transfer. This example illustrates the **write** routine. Assuming the device can only transfer **XX_MAXBUFSZ** bytes at a time, we calculate the amount of data to copy (line 22) and use **rmalloc** to allocate some space from the map. The call to **rmalloc** is protected against interrupts (line 23) from the device that may result in freeing map space. This way, if space is freed, we won't miss the corresponding **wakeup**(D3).

If the appropriate space cannot be allocated, we use **rmsetwant**(D3) to indicate that we want space (line 25) and then we sleep until a buffer is available. When a buffer becomes available, **rmfree** is called to return the space to the map and to wake the sleeping process. Then the call to **rmalloc** will succeed and the driver can then transfer data.

```
 1  #define XX_MAPSIZE     35
 2  #define XX_MEMSIZE     (16*1024)
 3  #define XX_MAXBUFSZ    1024

 4  struct map xx_map[XX_MAPSIZE];
    ...
 5  xx_start()
 6  {
 7    caddr_t bp;

 8    if ((bp = kmem_alloc(XX_MEMSIZE, KM_NOSLEEP)) == 0)
 9         cmn_err(CE_PANIC, "xx_start: could not allocate %d bytes",
10        XX_MEMSIZE);
11    rminit(xx_map, XX_MAPSIZE);
12    rmfree(xx_map, XX_MEMSIZE, bp);
13  }
    ...
14  xx_write(dev, uiop, crp)
15    dev_t dev;
16    uio_t *uiop;
17    cred_t *crp;
18  {
19    caddr_t addr;
20    size_t size;
21    int s;
    ...
22    while (uiop->uio_resid > 0) {
23        size = min(uiop->uio_resid, XX_MAXBUFSZ);
24        s = spl4();
25        while ((addr = (caddr_t)rmalloc(xx_map, size)) == NULL) {
26            rmsetwant(xx_map);
27            sleep((caddr_t)xx_map, PZERO);
28        }
29        splx(s);
          ...
30    }
    ...
```

On systems where the **rmallocmap** function is available, line 4 could become:

```
struct map *xx_map;
```

and line 10 could become:

```
if ((mp=rmallocmap(xx_MAPSIZE) == 0
      cmn_err (CE_PANIC, "xx_start: could not allocate map");
```

REFERENCES

rmalloc_wait(D3), **rmallocmap**(D3), **rmfree**(D3), **rmfreemap**(D3), **rminit**(D3), **rmsetwant**(D3)

NAME

`rmallocmap` – allocate and initialize a private space management map

SYNOPSIS

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

struct map *rmallocmap(ulong_t mapsize);
```

Arguments

mapsize	Number of entries for the map.

DESCRIPTION

`rmallocmap` allocates and initializes a private map array that can be used for the allocation of space.

Return Values

On success, **`rmallocmap`** returns a pointer to the newly allocated map. On failure, a **`NULL`** pointer is returned.

USAGE

Although **`rmallocmap`** allocates and initializes the map array itself, it does not allocate the space that the map will manage. This space must be allocated separately and must be added to the map using **`rmfree`**(D3) prior to attempting to allocate space from the map using **`rmalloc`**(D3) or **`rmalloc_wait`**(D3).

The system maintains the map list structure by size and index. The caller places whatever semantics on the units of size are appropriate for the type of space being managed. For example, units may be byte addresses, pages of memory, or blocks.

On systems where the **`rmallocmap`** function is available, DDI/DKI conforming drivers may only use **`map`** structures which have been allocated and initialized using **`rmallocmap`**. Use of **`map`** structures which have been obtained by any other means is prohibited on such systems.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read/write locks, and sleep locks may be held across calls to this function.

REFERENCES

`rmalloc`(D3), **`rmalloc_wait`**(D3), **`rmfree`**(D3), **`rmfreemap`**(D3)

NAME

rmalloc_wait – allocate space from a private space management map

SYNOPSIS

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

ulong_t rmalloc_wait(struct map *mp, size_t size);
```

Arguments

mp	Pointer to map to resource map.
size	Number of units to allocate.

DESCRIPTION

rmalloc_wait allocates space from a private map previously allocated using **rmallocmap**(D3).

Return Values

rmalloc_wait returns the base of the allocated space.

USAGE

rmalloc_wait is identical to **rmalloc**(D3), except that a call to **rmalloc_wait** will sleep (uninterruptible by signals), if necessary, until space becomes available.

Space allocated using **rmalloc_wait** may be returned to the map using **rmfree**(D3).

Level

Base only.

Synchronization Constraints

May sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function, but the driver writer must be cautious to avoid deadlock between the process holding the lock and trying to acquire the resource and another process holding the resource and trying to acquire the lock.

REFERENCES

rmalloc(D3), **rmallocmap**(D3), **rmfree**(D3), **rmfreemap**(D3)

NAME

`rmfree` – free space into a private space management map

SYNOPSIS

```
#include <sys/types.h>
#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.
size	Number of units to free into the map.
index	Index of the first unit of the space being freed.

DESCRIPTION

`rmfree` releases space into the private space management map pointed to by *mp* and wakes up any processes that are waiting for space.

Return Values

None

USAGE

`rmfree` should be called to return space that had been allocated by a previous call to **`rmalloc`**(D3), in which case *index* is the value returned from the corresponding call to **`rmalloc`**. **`rmfree`** should also be called to add space to a newly allocated map prior to the first call to **`rmalloc`**, in which case *index* specifies the base of the space being added.

Both *size* and *index* are in arbitrary units. The driver using the map places whatever semantics on the units are appropriate for the type of space being managed. For example, units may be byte addresses, pages of memory, or blocks on a device.

If the space being returned is adjacent to other space in the map, **`rmfree`** will coalesce the adjacent fragments.

If the **`rmfree`** call causes the number of fragments in the map to exceed the number of map entries specified by **`rminit`**(D3) (for singlethreaded drivers) or **`rmallocmap`**(D3) (for multithreaded drivers) the following warning message is displayed on the console:

`WARNING: rmfree map overflow` *mp* **`lost`** *size* **`items at`** *index*

This implies that the driver should specify a larger number of map entries when initializing the map.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See **`rmalloc`**(D3) for an example of **`rmfree`**.

REFERENCES

`rmalloc`(D3), **`rmalloc_wait`**(D3), **`rmallocmap`**(D3), **`rmfreemap`**(D3), **`rminit`**(D3), **`rmsetwant`**(D3)

NAME

`rmfreemap` – free a private space management map

SYNOPSIS

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

void rmfreemap(struct map *mp);
```

Arguments

mp	Pointer to the map to be freed.

DESCRIPTION

`rmfreemap` frees the map pointed to by *mp*.

Return Values

None

USAGE

The **`map`** structure array pointed to by *mp* must have been previously allocated by a call to **`rmallocmap`**(D3).

Before freeing the map, the caller must ensure that nobody is using space managed by the map, and that nobody is waiting for space in the map.

Level

Initialization, Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`rmalloc`(D3), **`rmalloc_wait`**(D3), **`rmallocmap`**(D3), **`rmfree`**(D3)

NAME

rmvb – remove a message block from a message

SYNOPSIS

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

mblk_t *rmvb(mblk_t *mp, mblk_t *bp);
```

Arguments

mp	Pointer to the message from which a message block is to be removed.
bp	Pointer to the message block to be removed.

DESCRIPTION

rmvb removes the message block specified by *bp* from the message specified *mp* and returns a pointer to the altered message.

Return Values

On success, a pointer to the message (minus the removed block) is returned. If *bp* was the only block in the message before **rmvb** was called, **NULL** is returned. If the designated message block (*bp*) was not in the message, –1 is returned.

USAGE

The message block is not freed, merely removed from the message. It is the caller's responsibility to free the message block.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

This routine removes all zero-length **M_DATA** message blocks from the given message. For each message block in the message, we save the next message block (line 9). If the current message block is of type **M_DATA** and has no data in its buffer (lines 10–11), then we remove the message block from the message (line 12) and free it (line 13). In either case, we continue with the next message block (line 15), until we have checked every message block in the message.

```
 1  void
 2  xxclean(mp)
 3    mblk_t *mp;
 4  {
 5    mblk_t *tmp;
 6    mblk_t *nmp;

 7    tmp = mp;
 8    while (tmp) {
      nmp = tmp->b_next;
      if ((tmp->b_datap->db_type == M_DATA) &&
         (tmp->b_rptr == tmp->b_wptr)) {
          mp = rmvb(mp, tmp);
          freeb(tmp);
      }
      tmp = nmp;
  }
}
```

NAME

rmvq – remove a message from a queue

SYNOPSIS

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

void rmvq(queue_t *q, mblk_t *mp);
```

Arguments

q	Pointer to the queue containing the message to be removed.
mp	Pointer to the message to remove.

DESCRIPTION

rmvq removes the message specified by *mp* from the queue specified by *q*.

Return Values

None

USAGE

A message can be removed from anywhere in a queue. To prevent modules and drivers from having to deal with the internals of message linkage on a queue, either **rmvq** or **getq**(D3) should be used to remove a message from a queue.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Warnings

mp must point to an existing message in the queue pointed to by *q*, or a system panic will occur.

Examples

See **insq**(D3) for an example of **rmvq**.

REFERENCES

getq(D3), **insq**(D3),

NAME

SAMESTR – test if next queue is of the same type

SYNOPSIS

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

int SAMESTR(queue_t *q);
```

Arguments

q	Pointer to the queue.

DESCRIPTION

The **SAMESTR** function checks whether the next queue in a stream (if it exists) is of the same type as the current queue (that is, both are read queues or both are write queues).

Return Values

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

USAGE

This function can be used to determine the point in a STREAMS-based pipe where a read queue is linked to a write queue.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

See the **put**(D2) manual page for an example of **SAMESTR**.

REFERENCES

OTHERQ(D3)

NAME

scsi_alloc – allocate communication channel between host adapter driver and a kernel level SCSI device driver

SYNOPSIS

```
#include "sys/types.h"
#include "sys/scsi.h"

int (*scsi_alloc[])(u_char adapter, u_char target, u_char lun,
                int option, void (*callback_function)(char *));
```

DESCRIPTION

A kernel level SCSI device driver calls **scsi_alloc** to initialize a communication connection between itself and a host adapter driver, in preparation for issuing SCSI commands. The *adapter*, *target*, and *lun* arguments specify the device. The option argument currently has two fields: *SCSIALLOC_EXCLUSIVE* indicates that the device driver wishes exclusive communication with the device.

SCSIALLOC_QDEPTH is an 8- bit mask which specifies the number of commands that the device driver will want to queue. It is advisory only and may be ignored by the host adapter driver. The *callback_function* argument may be useful for drivers that don’t use the *SCSIALLOC_EXCLUSIVE* option. When it is non-*NULL*, it specifies a function to call whenever there is sense data from device. It can be useful when more than one device driver will talk to one SCSI device, but one of the drivers n eeds to know about things like media removals. Only one device driver may specify a callback_function.

Return Values

scsi_alloc returns 0 if a communication connection could not be established, or the arguments are out of range, or the device was already allocate in exclusive use mode, or this request was for exclusive use and the device is already allocated (including possibly earlier requests by the same driver). Otherwise, it will return a positive value.

See Also

scsi_info(D3X), **scsi_command**(D3X), */usr/include/sys/scsi.h*

Note

scsi_alloc and **scsi_free** are actually an array of pointers to functions, indexed by SCSI host adapter driver number. See the SCSI chapter of the *IRIX Device Driver Programming Guide* or */usr/include/sys/scsi.h* for more information on how to use this function.

NAME

`scsi_command` – issue a command to a SCSI device

SYNOPSIS

```
#include "sys/types.h"
#include "sys/scsi.h"

void (*scsi_command[])(struct scsi_request *req);
```

DESCRIPTION

`scsi_command` is used to issue commands to SCSI devices. The caller (a kernel level SCSI device driver) fills out a struct *scsi_request* and passes a pointer to the request to *scsi_command.* See the SCSI chapter of the *IRIX Device Driver Programming Guide*
and */usr/include/sys/scsi.h* for more information on how to fill out a *scsi_request.*

Return Values

Success or failure is indicated by fields in the *scsi_request* structure passed to **`scsi_command`**. See the SCSI chapter of the *IRIX Device Driver Programming Guide*
or */usr/include/sys/scsi.h* for more information on return values.

See Also

`scsi_info`(D3X), **`scsi_alloc`**(D3X), */usr/include/sys/scsi.h*

Note

`scsi_command` is actually an array of pointers to functions, indexed by SCSI host adapter driver number. See the SCSI chapter of the *IRIX Device Driver Programming Guide*
or */usr/include/sys/scsi.h* for more information on how to use this function.

Unlike earlier versions of IRIX, it is not possible to call **`scsi_command`** and have it suspend the caller until the **`scsi_command`** completes (via a null callback pointer). Instead, the caller must use semaphores, or the (deprecated) sleep / wakeup mechanism, and a callback routine. Calls with a null *sr_notify_* will immediately return with a failure indication set in the *sr_status* field.

scsi_free(D3X)

NAME

scsi_free – free communication channel between host adapter driver and a kernel level SCSI device driver

SYNOPSIS

```
#include "sys/types.h"
#include "sys/scsi.h"

void (*scsi_free[])(u_char adapter, u_char target, u_char lun,
          void (*callback_function)(char *));
```

DESCRIPTION

scsi_free is used to terminate a communication connection. The arguments are the same as **scsi_alloc**(D3X), except that option is not used.

Return Values

None

See Also

scsi_alloc(D3X), **scsi_info**(D3X), **scsi_command**(D3X), */usr/include/sys/scsi.h*

Note

scsi_alloc(D3X) and **scsi_free**(D3X) are actually an array of pointers to functions, indexed by SCSI host adapter driver number. See the SCSI chapter of the *IRIX Device Driver Programming Guide* or */usr/include/sys/scsi.h* for more information on how to use this function.

NAME

`scsi_info` – get information about a SCSI device

SYNOPSIS

```
#include "sys/types.h"
#include "sys/scsi.h"

struct scsi_target_info * (*scsi_info[])(u_char adapter, u_char target, u_char lun);
```

DESCRIPTION

`scsi_info` issues an Inquiry command to the given *adapter*, *target*, and *lun*, returning a pointer to a struct *scsi_target_info*. The adapter argument indicates which adapter (or controller) to use. The target can be any number from 0 to 15, though the ID number of the host adapter itself (by default 0 on Integral controllers and 7 on VME controllers) is not available.

Return Values

If the given device does not exist, or there is an error getting the data, or if the arguments are out of range, *NULL* is returned. Otherwise a pointer to a struct *scsi_target_info* is returned.

See Also

`scsi_alloc`(D3X), **`scsi_command`**(D3X), */usr/include/sys/scsi.h*

Note

`scsi_info` is actually an array of pointers to functions, indexed by SCSI host adapter driver number. See the SCSI chapter of the *IRIX Device Driver Programming Guide* or */usr/include/sys/scsi.h* for more information on how to use this function.

NAME

sgset – assign physical addresses to a vector of software scatter-gather registers

SYNOPSIS

```
#include "sys/types.h"
#include "sys/buf.h"
#include "sys/sg.h"

sgset(struct buf *bp, struct sg *vec, int maxvec, int *resid);
```

DESCRIPTION

sgset provides a utility to manage a software equivalent of scatter-gather registers for devices that do not implement them. Based on the information provided in the *buf* type structure pointed to by *bp*, this routine fills in *maxvec* entries of a scatter-gather vector *vec*. If the number of vectors required to perform the transfer exceeds *maxvec*, the contents of *resid* is set to the number of pages remaining.

The buffer must not be for mapped address (*B_PAGEIO* for **buf**(40) not set).

The scatter gather entries are formatted in the following structure, excerpted from *sys/sg.h*:

```
struct sg {
        unsigned long sg_ioaddr; /* physical addrs of page */
        unsigned long sg_bcount; /* byte count of transfer */
};
```

Return Values

The number of vector entries used.

NAME

sleep – suspend process execution pending occurrence of an event

SYNOPSIS

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

int sleep(caddr_t event, int priority);
```

Arguments

event	Kernel address signifying an event for which the caller wishes to wait.
priority	A hint to the scheduling policy as to the relative priority the caller wishes to be assigned while running in the kernel after waking up.

DESCRIPTION

sleep suspends execution of a process to await certain events such as reaching a known system state in hardware or software. For instance, when a process wants to read a device and no data are available, the driver may need to call **sleep** to wait for data to become available before returning. This causes the kernel to suspend execution of the process that called **sleep** and schedule another process. The process that called **sleep** can be resumed by a call to the **wakeup** function with the same *event* specified as that used to call **sleep**.

Return Values

sleep returns **0** if the caller woke up because of a call to **wakeup**, or if the caller was stopped by a job control signal and subsequently continued. If the sleep is interrupted by a signal that does not cause the process to be stopped and the *priority* argument includes the **PCATCH** flag, the **sleep** call returns a value of **1**. If the sleep is interrupted by a signal and the **PCATCH** flag is not set, the process will **longjmp** out of the driver and the **sleep** call will never return to the calling code.

USAGE

event Argument

The address has no significance except that the same address must be passed to **wakeup**(D3) to resume the sleeping process. The address used should be the address of a kernel data structure associated with the driver, or one of the driver's own data structures. Use of arbitrary addresses not associated with a private data structure can result in conflict with other, unrelated **sleep** and **wakeup** operations in the kernel.

priority Argument

In general, a lower value will result in more favorable scheduling although the exact semantic of the priority argument is specific to the scheduling class of the caller, and some scheduling classes may choose to ignore the argument for the purposes of assigning a scheduling priority.

In addition to the scheduling semantics, the value of the *priority* argument determines whether the sleep may be interrupted by signals. If the value of *priority* is less than or equal to the value of the constant **PZERO** (defined in **sys/param.h**), the sleeping process will not be awakened by a signal. If the value of *priority* is greater than **PZERO** and the **PCATCH** bit flag is **OR**ed into the *priority* argument, the process will wake up prematurely (without a call to **wakeup**) upon receipt of a non-ignored, non-held signal and will normally return **1** to the calling code. If *priority* is greater than **PZERO** and **PCATCH** is not set, the **sleep** function will **longjmp** out of the driver upon receipt of a signal and will never return to the caller.

General Considerations

If a process were to sleep while it is manipulating global data inside a critical section of driver code, it would be possible for another process to execute base level driver code which manipulates the same data while the first process was sleeping, resulting in data corruption. A driver should not sleep inside such a critical section unless it takes explicit steps to prevent concurrent access to the data (for example, the driver could implement its own locking protocol to protect the data).

The value for *priority* should be selected based on whether or not a **wakeup** is certain to occur as well as the importance of the driver and of any resources that the driver will hold after waking up. If the driver is holding or waiting for a critical kernel resource or is otherwise crucial to the performance of the system, and the corresponding call to **wakeup** is guaranteed to happen, the driver should specify a *priority* argument less than or equal to **PZERO**. If the driver is less performance critical or it is possible that the **wakeup** may not occur, the driver should specify a *priority* argument greater than **PZERO**.

If there is any driver state that needs to be cleaned up in the event of a signal, the driver should **OR** the **PCATCH** flag in with the priority argument. Typical items that need cleaning up are locked data structures that should be unlocked or dynamically allocated resources that need to be freed. When **PCATCH** is specified **sleep** will normally return a **1** in the event of a signal, indicating that the calling routine should perform any necessary cleanup and then return.

If **sleep** is called from the driver **strategy**(D2) routine, the caller should **OR** the *priority* argument with **PCATCH** or select a *priority* of **PZERO** or less.

Level

Base only.

Synchronization Constraints

Can sleep.

REFERENCES

wakeup(D3)

NAME

SLEEP_ALLOC – allocate and initialize a sleep lock

SYNOPSIS

```
#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>
```

sleep_t *SLEEP_ALLOC(int *arg*, **lkinfo_t ****lkinfop*, **int** *flag***);**

Arguments

arg	Reserved for future use (must be equal to zero).
lkinfop	Reserved for future use (must be equal to -1).
flag	Specifies whether the caller is willing to sleep waiting for memory.

DESCRIPTION

SLEEP_ALLOC dynamically allocates and initializes an instance of a sleep lock. The lock is initialized to the unlocked state.

If *flag* is set to **KM_SLEEP**, the caller will sleep if necessary until sufficient memory is available. If *flag* is set to **KM_NOSLEEP**, the caller will not sleep, but **SLEEP_ALLOC** will return **NULL** if sufficient memory is not immediately available.

Return Values

Upon successful completion, **SLEEP_ALLOC** returns a pointer to the newly allocated lock. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **SLEEP_ALLOC** returns a **NULL** pointer.

USAGE

Level

Base only if *flag* is set to **KM_SLEEP**.

Initialization, Base or Interrupt if *flag* is set to **KM_NOSLEEP**.

Synchronization Constraints

May sleep if flag is set to **KM_SLEEP**.

Driver-defined basic locks and read / write locks may be held across calls to this function if *flag* is **KM_NOSLEEP** but may not be held if *flag* is **KM_SLEEP**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of *flag*.

REFERENCES

initnsema(D3X), **initnsema_mutex**(D3X), **SLEEP_DEALLOC**(D3), **SLEEP_LOCK**(D3), **SLEEP_LOCK_SIG**(D3), **SLEEP_LOCKAVAIL**(D3), **SLEEP_TRYLOCK**(D3), **SLEEP_UNLOCK**(D3)

SLEEP_DEALLOC(D3)

NAME

SLEEP_DEALLOC – deallocate an instance of a sleep lock

SYNOPSIS

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

void SLEEP_DEALLOC(sleep_t *lockp);
```

Arguments

lockp	Pointer to the sleep lock to be deallocated.

DESCRIPTION

SLEEP_DEALLOC deallocates the lock specified by *lockp*.

Return Values

None

USAGE

Attempting to deallocate a lock that is currently locked or is being waited for is an error and results in undefined behavior.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks (other than the one being deallocated), may be held across calls to this function.

REFERENCES

freesema(D3X), **SLEEP_ALLOC**(D3), **SLEEP_LOCK**(D3), **SLEEP_LOCK_SIG**(D3), **SLEEP_LOCKAVAIL**(D3), **SLEEP_TRYLOCK**(D3), **SLEEP_UNLOCK**(D3)

NAME

SLEEP_LOCK – acquire a sleep lock

SYNOPSIS

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

void SLEEP_LOCK(sleep_t *lockp, int priority);
```

Arguments

lockp	Pointer to the sleep lock to be acquired.
priority	Reserved for future use (must be equal to -1).

DESCRIPTION

SLEEP_LOCK acquires the sleep lock specified by *lockp*. If the lock is not immediately available, the caller is put to sleep (the caller's execution is suspended and other processes may be scheduled) until the lock becomes available to the caller, at which point the caller wakes up and returns with the lock held.

The caller will not be interrupted by signals while sleeping inside **SLEEP_LOCK**.

Return Values

None

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function subject to the recursion restrictions described below.

Warnings

Sleep locks are not recursive. A call to **SLEEP_LOCK** attempting to acquire a lock that is currently held by the calling context will result in deadlock.

REFERENCES

psema(D3X), **SLEEP_ALLOC**(D3), **SLEEP_DEALLOC**(D3), **SLEEP_LOCK_SIG**(D3), **SLEEP_LOCKAVAIL**(D3), **SLEEP_TRYLOCK**(D3), **SLEEP_UNLOCK**(D3)

SLEEP_LOCKAVAIL(D3)

NAME

SLEEP_LOCKAVAIL – query whether a sleep lock is available

SYNOPSIS

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

boolean_t SLEEP_LOCKAVAIL(sleep_t *lockp);
```

Arguments

lockp Pointer to the sleep lock to be queried.

DESCRIPTION

SLEEP_LOCKAVAIL returns an indication of whether the sleep lock specified by *lockp* is currently available.

Return Values

SLEEP_LOCKAVAIL returns **TRUE** (a non-zero value) if the lock was available or **FALSE** (zero) if the lock was not available.

USAGE

The state of the lock may change and the value returned may no longer be valid by the time the caller sees it. The caller is expected to understand that this is ‘‘stale data’’ and is either using it as a heuristic or has arranged for the return value to be meaningful by other means.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

SLEEP_ALLOC(D3), **SLEEP_DEALLOC**(D3), **SLEEP_LOCK**(D3), **SLEEP_LOCK_SIG**(D3), **SLEEP_TRYLOCK**(D3), **SLEEP_UNLOCK**(D3)

NAME

SLEEP_LOCK_SIG – acquire a sleep lock

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>
#include <sys/param.h>
```

boolean_t SLEEP_LOCK_SIG(sleep_t ****lockp*, int** *priority***);**

Arguments

lockp	Pointer to the sleep lock to be acquired.
priority	A hint to the scheduling policy as to the relative priority the caller wishes to be assigned while running in the kernel after waking up.

DESCRIPTION

SLEEP_LOCK_SIG acquires the sleep lock specified by *lockp*. If the lock is not immediately available, the caller is put to sleep (the caller's execution is suspended and other processes may be scheduled) until the lock becomes available to the caller, at which point the caller wakes up and returns with the lock held.

SLEEP_LOCK_SIG may be interrupted by a signal, in which case it may return early without acquiring the lock.

If the function is interrupted by a job control stop signal (e.g., **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, **SIGTTOU**) which results in the caller entering a stopped state, the **SLEEP_LOCK_SIG** function will transparently retry the lock operation upon continuing (the call will not return without the lock).

If the function is interrupted by a signal other than a job control stop signal, or by a job control stop signal that does not result in the caller stopping (because the signal has a non-default disposition), the **SLEEP_LOCK_SIG** call will return early without acquiring the lock.

Return Values

SLEEP_LOCK_SIG returns **TRUE** (a non-zero value) if the lock is successfully acquired or **FALSE** (zero) if the function returned early because of a signal.

USAGE

priority Argument

In general, a lower value will result in more favorable scheduling although the exact semantic of the priority argument is specific to the scheduling class of the caller, and some scheduling classes may choose to ignore the argument for the purposes of assigning a scheduling priority.

The value of priority must be greater than PZERO (defined in sys/param.h)

In general, a higher relative priority should be used when the caller is attempting to acquire a highly contended lock or resource,or when the caller is already holding one or more locks or kernel resources upon entry to **SLEEP_LOCK_SIG**.

Level

Base only.

Synchronization Constraints

Can sleep.

Driver-defined basic locks and read / write locks may not be held across calls to this function.

Driver-defined sleep locks may be held across calls to this function subject to the recursion restrictions described below.

Warnings

Sleep locks are not recursive. A call to **`SLEEP_LOCK_SIG`** attempting to acquire a lock that is currently held by the calling context will result in deadlock.

REFERENCES

`SLEEP_ALLOC`(D3), **`SLEEP_DEALLOC`**(D3), **`SLEEP_LOCK`**(D3), **`SLEEP_LOCKAVAIL`**(D3), **`SLEEP_TRYLOCK`**(D3), **`SLEEP_UNLOCK`**(D3), **`signals`**(D5), **`psema`**(D3X)

NAME

SLEEP_TRYLOCK – try to acquire a sleep lock

SYNOPSIS

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

boolean_t SLEEP_TRYLOCK(sleep_t ****lockp*);**

Arguments

lockp	Pointer to the sleep lock to be acquired.

DESCRIPTION

If the lock specified by *lockp* is immediately available (can be acquired without sleeping) the **SLEEP_TRYLOCK** function acquires the lock. If the lock is not immediately available, **SLEEP_TRYLOCK** returns without acquiring the lock.

Return Values

SLEEP_TRYLOCK returns **TRUE** (a non-zero value) if the lock is successfully acquired or **FALSE** (zero) if the lock is not acquired.

USAGE

Level

Base only.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

cpsema(D3X), **cvsema**(D3X), **SLEEP_ALLOC**(D3), **SLEEP_DEALLOC**(D3), **SLEEP_LOCK**(D3), **SLEEP_LOCK_SIG**(D3), **SLEEP_LOCKAVAIL**(D3), **SLEEP_UNLOCK**(D3)

NAME

SLEEP_UNLOCK – release a sleep lock

SYNOPSIS

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

void SLEEP_UNLOCK(sleep_t *lockp);
```

Arguments

lockp	Pointer to the sleep lock to be released.

DESCRIPTION

SLEEP_UNLOCK releases the sleep lock specified by *lockp*. If there are processes waiting for the lock, one of the waiting processes is awakened.

Return Values

None

USAGE

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

SLEEP_ALLOC(D3), **SLEEP_DEALLOC**(D3), **SLEEP_LOCK**(D3), **SLEEP_LOCK_SIG**(D3), **SLEEP_LOCKAVAIL**(D3), **SLEEP_TRYLOCK**(D3), **vsema**(D3X)

NAME

spl – block / allow interrupts on a processor

SYNOPSIS

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

int splbase(void);
int spltimeout(void);
int spldisk(void);
int splstr(void);
int spltty(void);
int splhi(void);
int spl0(void);
int spl7(void);

void splx(int oldlevel);
```

Arguments

oldlevel	Last set priority value (only **splx** has an input argument).

DESCRIPTION

The **spl** functions block or allow servicing of interrupts on the processor on which the function is called.

Return Values

All **spl** functions except **splx** return the previous priority level for use by **splx**.

USAGE

Hardware devices are assigned to interrupt priority levels depending on the type of device. Each **spl** function which blocks interrupts is associated with some machine dependent interrupt priority level and will prevent interrupts occurring at or below this priority level from being serviced on the processor on which the **spl** function is called.

On a multiprocessor system, interrupts may be serviced by more than one processor and, therefore, use of a **spl** function alone is not sufficient to prevent interrupt code from executing and manipulating driver data structures during a critical section. Drivers that must prevent execution of interrupt-level code in order to protect the integrity of their data should use basic locks for this purpose [see **LOCK_ALLOC**(D3)].

The **spl** functions include the following:

splbase	Block no interrupts. Should only be used by base-level code that knows it is not nested within any section of protected critical code.
spltimeout	Block functions scheduled by **itimeout** and **dtimeout**.
spldisk	Block disk device interrupts.
splstr	Block STREAMS interrupts.

spltty	Used by a TTY driver to protect critical code. **spltty** is mapped to **splstr**.
splhi	Block all interrupts. Can be used in any type of driver to mask out all interrupts including the clock, and should be used very sparingly.
spl0	Equivalent to **splbase**.
spl7	Equivalent to **splhi**.

To ensure driver portability, the named **spl** functions (such as **splbase** or **spltimeout**) should be used whenever possible. The numbered **spl** functions (**spl0** and **spl7**) should be used only when an interrupt priority level must be set to a specific value.

Calling a given **spl** function will block interrupts specified for that function as well as interrupts at equal and lower levels. The notion of low vs. high levels assumes a defined order of priority levels. The following partial order is defined:

```
    splbase <= spltimeout <= spldisk,splstr <= splhi
```

The ordering of **spldisk** and **splstr** relative to each other is not defined.

When setting a given priority level, the previous level returned should be saved and **splx** or **UNLOCK**(D3) should be used as appropriate to restore this level.

Interrupt-level code must never lower the interrupt priority level below the level at which the interrupt handler was entered. For example, if an interrupt handler is entered at the priority level associated with **spldisk**, the handler must not call **spltimeout**.

Level

Base or Interrupt.

Synchronization Considerations

All **spl** functions do not sleep.

Driver-defined basic locks and read / write locks may be held across calls to these functions, but the **spl** call must not cause the priority level to be lowered below the level associated with the lock.

Driver-defined sleep locks may be held across calls to these functions.

REFERENCES

LOCK(D3), **LOCK_ALLOC**(D3),

NAME

strcat – concatenate strings

SYNOPSIS

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

char *strcat(char ****sptr1*, const char ****sptr2***)**

Arguments

The arguments *sptr1* and *sptr2* each point to strings, and each string is an array of characters terminated by a null-character.

DESCRIPTION

The function **strcat** appends a copy of the string pointed to by *sptr2* including the terminating null-character to the end of the string pointed to by *sptr1*. The initial character in the string pointed to by *sptr2* replaces the null-character at the end of the string pointed to by *sptr1*.

The function **strcat** alters *sptr1* without checking for overflow of the array pointed to by *sptr1*. If copying takes place between strings that overlap, the behavior is undefined.

Return Values

The function **strcat** returns the value of *sptr1*, which points to the null-terminated result.

USAGE

Character movement is performed differently in different implementations; thus, overlapping moves may yield surprises.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

bcopy(D3)

NAME

strcpy – copy a string

SYNOPSIS

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

char *strcpy (char *sptr1, const char *sptr2)
```

Arguments

The arguments *sptr1* and *sptr2* each point to strings, and each string is an array of characters terminated by a null-character.

DESCRIPTION

The function **strcpy** copies the string pointed to by *sptr2* (including the terminating null-character) into the string pointed to by *sptr1*, stopping after the null-character has been copied, and returns the string pointed to by *sptr1*.

The function **strcpy** alters *sptr1* without checking for overflow of the string pointed to by *sptr1*. If copying takes place between strings that overlap, the behavior is undefined.

Return Values

The function **strcpy** returns the value of *sptr1*, which points to the null-terminated result.

USAGE

Character movement is performed differently in different implementations; thus, overlapping moves may yield surprises.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

bcopy(D3)

NAME

`streams_interrupt` – synchronize interrupt-level function with STREAMS mechanism

SYNOPSIS

`#include <strmp.h>`

`typedef void (*strintrfunc_t)(void *, void *, void *);`

`int streams_interrupt(strintrfunc_t` *func*, **`void *`***a1*, **`void *`***a2*, **`void *`***a3***`);`**

DESCRIPTION

`streams_interrupt` provides writers of STREAMS-based device drivers with an interface for synchronizing interrupt-level functions with the STREAMS mechanism on multi-processor IRIX systems. Under IRIX, it is not permitted to call STREAMS interface routines (e.g., **`allocb`**(), **`putq`**(), **`qenable`**()) or otherwise manipulate STREAMS data structures from interrupt level without first synchronizing with the underlying STREAMS mechanism.

Failure to properly synchronize could result in corrupted data structures and kernel panics.

`streams_interrupt` attempts to synchronize with the STREAMS mechanism and execute *func*, before returning to the caller. If **`streams_interrupt`** cannot immediately synchronize with the STREAMS mechanism, it will schedule *func* for execution the next time synchronization can be achieved and will return to the caller. Since the time between calling **`streams_interrupt`** and the time that *func* is executed is indeterminate, it is not advisable to use *func* to perform time-critica l tasks (e.g., resetting hardware state, reading hardware data buffers, etc.).

`streams_interrupt` does not guarantee the spl level that *func* will run at. It is the responsibility of the driver writer to explicitly set the desired spl level within *func*. However, the driver writer "must not" under any circumstances set the spl level to spl0 within *func*.

Diagnostics

`streams_interrupt` returns 1 if *func* was executed, 0 if *func* was scheduled for later execution, and –1 on error.

See Also

`STREAMS_TIMEOUT`(D3X), **`untimeout`**(D3), *IRIX Device Driver Programming Guide*

NAME

STREAMS_TIMEOUT – synchronize timeout with STREAMS mechanism

SYNOPSIS

#include <strmp.h>

toid_t STREAMS_TIMEOUT(strtimeoutfunc_t *func*, **void** **arg*, **int** *time***);**
toid_t STREAMS_TIMEOUT1(strtimeoutfunc_t *func*, **void** **arg*, **int** *time*, **void** **arg1***);**
toid_t STREAMS_TIMEOUT2(strtimeoutfunc_t *func*, **void** **arg*, **int** *time*, **void** **arg1*, **void** **arg2***);**

DESCRIPTION

The **STREAMS_TIMEOUT** macros provide writers of STREAMS device drivers and modules with an interface for synchronizing timeouts with the STREAMS mechanism on multi-processor IRIX systems. Under IRIX, it is not permitted to call STREAMS interface routines (e.g., **allocb**(), **putq**(), **qenable**()) or otherwise manipulate STREAMS data structures from a function called via a timeout without first synchronizing with the underlying STREAMS mechanism.

Failure to properly synchronize could result in corrupted data structures and kernel panics.

The **STREAMS_TIMEOUT** interfaces arrange for *func* to be called in *time* / HZ seconds, and guarantee that it will be properly synchronized with the STREAMS mechanism.

One, two or three arguments may be passed to *func* by using *STREAMS_TIMEOUT*, *STREAMS_TIMEOUT1* or *STREAMS_TIMEOUT2* respectively.

The **STREAMS_TIMEOUT** interfaces do not guarantee the spl level that *func* will run at. It is the responsibility of the driver or module writer to explicitly set the desired spl level within *func*. However, the driver or module writer "must not" under any circumstances set the spl level to spl0 within *func*.

Diagnostics

The **STREAMS_TIMEOUT** interfaces return a positive *toid_t* on success. This *toid_t* value may be used by a subsequent call to **untimeout**() to cancel the timeout. If an error is detected while setting the timeout, a *toid_t* of 0 will be returned and no timeout will be set.

See Also

streams_interrupt(D3X), **untimeout**(D3), *IRIX Device Driver Programming Guide*

NAME

`strlog` – submit messages to the **`log`** driver

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/log.h>
#include <sys/ddi.h>

int strlog(short mid, short sid, char level, ushort_t flags,
      char *fmt, ... /* args */);
```

Arguments

mid	Identification number of the module or driver submitting the message.
sid	Identification number for a particular minor device.
level	Tracing level for selective screening of low priority messages.
flags	Bitmask of flags indicating message purpose.
fmt	**`printf`**(3S) style format string.
args	Zero or more arguments to **`printf`**.

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.

USAGE

flags Argument

Valid values for *flags* are:

`SL_ERROR`	Message is for error logger.
`SL_TRACE`	Message is for tracing.
`SL_CONSOLE`	Message is for console logger.
`SL_NOTIFY`	If **`SL_ERROR`** is also set, mail copy of message to system administrator.
`SL_FATAL`	Modifier indicating error is fatal.

`SL_WARN`	Modifier indicating error is a warning.
`SL_NOTE`	Modifier indicating error is a notice.

fmt Argument

The **`%s`**, **`%e`**, **`%g`**, and **`%G`** formats are not allowed.

printf args

`args` can specify a maximum of **`NLOGARGS`**, currently three.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

`log`(7), **`strace`**(1M), **`strerr`**(1M)

NAME

strqget – get information about a queue or band of the queue

SYNOPSIS

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

int strqget(queue_t *q, qfields_t what, uchar_t pri, long *valp);
```

Arguments

q	Pointer to the queue.
what	The field of the queue about which to return information.
pri	Priority band of the queue about which to obtain information.
valp	Pointer to the memory location where the value is to be stored.

DESCRIPTION

strqget gives drivers and modules a way to get information about a queue or a particular priority band of a queue without directly accessing STREAMS data structures.

Return Values

On success, **0** is returned. An error number is returned on failure. The actual value of the requested field is returned through the reference parameter, *valp*.

USAGE

Valid *what* values are:

QHIWAT	High water mark of the specified priority band.
QLOWAT	Low water mark of the specified priority band.
QMAXPSZ	Maximum packet size of the specified priority band.
QMINPSZ	Minimum packet size of the specified priority band.
QCOUNT	Number of bytes of data in messages in the specified priority band.
QFIRST	Pointer to the first message in the specified priority band.
QLAST	Pointer to the last message in the specified priority band.
QFLAG	Flags for the specified priority band [see **queue**(D4)].

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

queue(D4), **strqset**(D3),

NAME

strqset – change information about a queue or band of the queue

SYNOPSIS

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

int strqset(queue_t *q, qfields_t what, uchar_t pri, long val);
```

Arguments

q	Pointer to the queue.
what	The field of the queue to change.
pri	Priority band of the queue to be changed.
val	New value for the field to be changed.

DESCRIPTION

strqset gives drivers and modules a way to change information about a queue or a particular priority band of a queue without directly accessing STREAMS data structures.

Return Values

On success, **0** is returned. An error number is returned on failure.

USAGE

Valid values for *what* are:

QHIWAT	High water mark of the specified priority band.
QLOWAT	Low water mark of the specified priority band.
QMAXPSZ	Maximum packet size of the specified priority band.
QMINPSZ	Minimum packet size of the specified priority band.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

queue(D4), **strqget**(D3)

NAME

subyte – set (write) a byte to user space

SYNOPSIS

int subyte(char ****usr_v_addr*, char** *c***);**

DESCRIPTION

subyte writes the given (8-bit) byte, *c*, to the specified address, *user_v_addr*, in the currently mapped user process' address space.

Return Values

Upon successful completion, **subyte** returns 0. Otherwise, **subyte** returns –1, indicating an invalid user virtual address.

See Also

copyout(D3), **fubyte**(D3X), **suword**(D3X),

NAME

`suword` – set (write) a word to user space

SYNOPSIS

`int suword(int *`*usr_v_addr***`, int`** *i***`);`**

Arguments

usr_v_addr
: Specified address in the currently mapped user process' address space.

i
: 32-bit word.

DESCRIPTION

`suword` writes the given (32-bit) word, i, to the specified address, user_v_addr, in the currently mapped user process' address space.

Return Values

Upon successful completion, **`suword`** returns 0. Otherwise, **`suword`** returns –1, indicating an invalid user virtual address.

See Also

`fuword`(D3X), **`subyte`**(D3X), **`copyout`**(D3)

NAME

TRYLOCK – try to acquire a basic lock

SYNOPSIS

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

int TRYLOCK(lock_t *lockp, pl_t pl);
```

Arguments

lockp	Pointer to the basic lock to be acquired.
pl	The interrupt priority level to be set while the lock is held by the caller.

DESCRIPTION

If the lock specified by *lockp* is immediately available (can be acquired without waiting) **TRYLOCK** sets the interrupt priority level in accordance with the value specified by *pl* and acquires the lock. If the lock is not immediately available, the function returns without acquiring the lock.

Return Values

If the lock is acquired, **TRYLOCK** returns the previous interrupt priority level for use by UNLOCK. If the lock is not acquired the value **invpl** is returned.

USAGE

Because some implementations require that interrupts that might attempt to acquire the lock be blocked on the processor on which the lock is held, portable drivers must specify a *pl* value that is sufficient to block out any interrupt handler that might attempt to acquire this lock. See the description of the *min_pl* argument to **LOCK_ALLOC**(D3) for additional discussion and a list of the valid values for *pl*.

TRYLOCK may be used to acquire a lock in a different order from the order defined by the lock hierarchy.

When called from interrupt level, the *pl* argument must not specify a priority level below the level at which the interrupt handler is running.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

LOCK(D3), **LOCK_ALLOC**(D3), **LOCK_DEALLOC**(D3), **UNLOCK**(D3)

NAME

uiomove – copy data using **uio**(D4) structure

SYNOPSIS

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

int uiomove(caddr_t addr, long nbytes, uio_rw_t rwflag,
                   uio_t *uiop);
```

Arguments

addr	Source / destination kernel address of the copy.
nbytes	Number of bytes to copy.
rwflag	Flag indicating read or write operation. Valid values are **UIO_READ** and **UIO_WRITE**.
uiop	Pointer to the **uio** structure for the copy.

DESCRIPTION

The **uiomove** function copies *nbytes* of data between the kernel address *addr* and the space defined by the **uio** structure pointed to by *uiop*. If *rwflag* is **UIO_READ**, the data is copied from *addr* to the space described by the **uio** structure. If *rwflag* is **UIO_WRITE**, the data is copied from the space described by the **uio** structure to *addr*.

The **uio_segflg** member of the **uio** structure specifies the type of space described by the **uio** structure. If **uio_segflg** is set to **UIO_SYSSPACE** the **uio** structure describes a portion of the kernel address space. If **uio_segflg** is set to **UIO_USERSPACE** the **uio** structure describes a portion of the user address space.

If the copy is successful, **uiomove** updates the appropriate members of the **uio** and **iovec**(D4) structures to reflect the copy (**uio_offset** and **iov_base** are increased by *nbytes* and **uio_resid** and **iov_len** are decrease by *nbytes*).

Return Values

uiomove returns 0 on success or an error number on failure.

USAGE

Level

Base only if **uio_segflg** is set to **UIO_USERSPACE**.

Base or Interrupt if **uio_segflg** is set to **UIO_SYSSPACE**.

Synchronization Constraints

May sleep if **uio_segflg** is set to **UIO_USERSPACE**.

Driver-defined basic locks and read / write locks may be held across calls to this function if **uio_segflg** is **UIO_SYSSPACE** but may not be held if **uio_segflg** is **UIO_USERSPACE**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of **uio_segflg**.

Warnings

If *addr* specifies an address in user space or if the value of **uio_segflg** is not consistent with the type of address space described by the **uio** structure, the system can panic.

When holding locks across calls to this function, multithreaded drivers must be careful to avoid creating a deadlock. During the data transfer, page fault resolution might result in another I/O to the same device. For example, this could occur if the driver controls the disk drive used as the swap device.

REFERENCES

bcopy(D3), **copyin**(D3), **copyout**(D3), **ureadc**(D3), **uwritec**(D3), **iovec**(D4), **uio**(D4)

NAME

uiophysio – set up user data space for I/O

SYNOPSIS

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

int uiophysio(int (*strat)(struct buf *), struct buf *bp, dev_t dev, int rwflag, str
```

Arguments

strat	Address of the driver routine.
bp	Pointer to the **buf**(D4) structure describing the I/O request.
dev	External device number.
rwflag	Flag indicating whether the access is a read or a write.
uiop	Pointer to the **uio**(D4) structure that defines the user space of the I/O request.

DESCRIPTION

uiophysio prepares the user's address space for DMA I/O and encapsulates the transfer information in a buffer header.

Return Values

uiophysio returns 0 if the result is successful, or the appropriate error number on failure. If a partial transfer occurs, the *uio* structure is updated to indicate the amount not transferred and an error is returned. **uiophysio** returns the *ENOSPC* error if an attempt is made to read beyond the end of the device. If a read is performed at the end of the device, 0 is returned. *ENOSPC* is also returned if an attempt is made to write at or beyond the end of a the device. *EFAULT* is returned if user memory is not valid. *EAGAIN* is returned if **uiophysio** could not lock all of the pages.

USAGE

uiophysio performs the following functions:

- Sets up a buffer header describing the transfer; faults pages in and locks the pages impacted by the I/O transfer so they can't be swapped out
- Calls the routine named in the *strat* parameter, passing a pointer to a *buf* structure
- Sleeps until the transfer is complete and is awakened by a call to **biodone**(D3) from the driver's I/O completion handler
- Performs the necessary cleanup and updates, then returns to the driver routine

If *bp* is set to *NULL*, a buffer is allocated temporarily and freed after the transfer completes.

If *rwflag* is set to *B_READ*, the direction of the data transfer will be from the kernel or device to the user's buffer. If *rwflag* is set to *B_WRITE*, the direction of the data transfer will be from the user's buffer to the kernel or device.

Level

Base only.

See Also

buf(D4), **ioctl**(D2), **read**(D2), **strategy**(D2), **physiock**(D3), **uio**(D4), **write**(D2)

NAME

unbufcall – cancel a pending **bufcall** request

SYNOPSIS

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

void unbufcall(toid_t id);
```

Arguments

id	Non-zero identifier returned from a prior call to **bufcall**(D3) or **esbbcall**(D3).

DESCRIPTION

unbufcall cancels the pending **bufcall** or **esbbcall** request specified by *id*.

Return Values

None

USAGE

If **unbufcall** is called while any function called by the pending **bufcall** or **esbbcall** request is running, the call to **unbufcall** has no effect.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function.

Example

See **bufcall** for the other half of this example.

In the module close routine, if a **bufcall** request is pending (line 14), we cancel it (line 15). Otherwise, if a **itimeout** request is pending (line 16), we cancel it (line 17). Then the **m_type** field in the module's private data structure is set to 0, indicating no pending **bufcall** or **itimeout**.

```
 1  struct mod {
 2     long   m_id;
 3     char   m_type;
       ...
 4  };
 5  #define TIMEOUT 1
 6  #define BUFCALL 2
    ...
 7  modclose(q, flag, crp)
 8     queue_t *q;
 9     int flag;
10     cred_t *crp;
11  {
12     struct mod *modp;

13     modp = (struct mod *)q->q_ptr;
```

```
    if (modp->m_type == BUFCALL)
            unbufcall(modp->m_id);
    else if (modp->m_type == TIMEOUT)
            untimeout(modp->m_id);
    modp->m_type = 0;
    ...
```

REFERENCES

bufcall(D3), **esbbcall**(D3)

NAME

undma – unlock physical memory in user space

SYNOPSIS

```
#include "sys/types.h"
#include "sys/buf.h"

undma(void *usr_v_addr, unsigned int num_bytes, int rw);
```

Arguments

usr_v_addr
User process address space.

num_bytes
Number of bytes.

rw *B_READ* or *B_WRITE* (should match corresponding **userdma**() call).

DESCRIPTION

userdma prepares memory before a DMA operation into or from a user process' address space and invalidates the data cache lines corresponding to the given address and count, if necessary.

When the operation is complete, call **undma** to unlock these pages.

Return Values

None

NAME

unlinkb – remove a message block from the head of a message

SYNOPSIS

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

mblk_t *unlinkb(mblk_t *mp);
```

Arguments

mp	Pointer to the message.

DESCRIPTION

unlinkb removes the first message block from the message pointed to by *mp*.

Return Values

unlinkb returns a pointer to the remainder of the message after the first message block has been removed. If there is only one message block in the message, **NULL** is returned.

USAGE

The removed message block is not freed. It is the caller’s responsibility to free it.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

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);
}
```

REFERENCES

linkb(D3)

NAME

UNLOCK – release a basic lock

SYNOPSIS

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

void UNLOCK(lock_t *lockp, int pl);
```

Arguments

lockp	Pointer to the basic lock to be released.
pl	The interrupt priority level to be set after releasing the lock. This argument should be the value returned by **LOCK**.

DESCRIPTION

UNLOCK releases the basic lock specified by *lockp* and then sets the interrupt priority level in accordance with the value specified by *pl*.

Return Values

None

USAGE

See the description of the *min_pl* argument to **LOCK_ALLOC**(D3) for a list of the valid values for *pl*. If lock calls are not being nested or if the caller is unlocking in the reverse order that locks were acquired, the *pl* argument should be the value that was returned from the corresponding call to acquire the lock. The caller may need to specify a different returned value by other **LOCK** for *pl* if nested locks are released in some order other than the reverse order of acquisition, so as to ensure that the interrupt priority level is kept sufficiently high to block interrupt code that might attempt to acquire locks which are still held.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

REFERENCES

LOCK(D3), **LOCK_ALLOC**(D3), **LOCK_DEALLOC**(D3), **TRYLOCK**(D3)

NAME

untimeout – cancel previous timeout request

SYNOPSIS

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

void untimeout(toid_t id);
```

Arguments

id	Non-zero identifier returned from a prior call to **dtimeout**(D3) or **itimeout**(D3).

DESCRIPTION

untimeout cancels the pending timeout request specified by *id*.

Return Values

None

USAGE

On uniprocessor systems, if **untimeout** is called while any function called by the pending timeout request is running, then there is no effect.

On multiprocessor systems, if **untimeout** is called while any function called by the pending timeout request is running, **untimeout** will not return until the function completes.

Note that any function that runs as a result of a call to **itimeout** (or to **dtimeout**) cannot use **untimeout** to cancel itself.

Level

Base or Interrupt, with the following exception on multiprocessor systems: For **itimeout**(D3) and **dtimeout**(D3), the **untimeout** can only be performed from interrupt levels less than, or equal to, the level specified when the function was scheduled.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may not be held across calls to this function if these locks are contended by the function being canceled.

Examples

See **unbufcall**(D3) for an example of **untimeout**.

REFERENCES

delay(D3), **dtimeout**(D3), **itimeout**(D3), **unbufcall**(D3)

NAME

untimeout_func – cancel a previous invocation of **timeout** by function

SYNOPSIS

#include "sys/types.h"

untimeout_func(int (**function***)(), caddr_t** *arg***);**

DESCRIPTION

untimeout_func works much like untimeout in that **untimeout_func** cancels a previous timeout scheduled by **itimeout**(D3). However, **untimeout_func** cancels a scheduled timeout that is identified by the *function* and first argument specified in the original **itimeout**(D3) call. If more than one call to the given function is scheduled with the same argument, **untimeout_func** cancels the timeout with the nearest expiration time.

Use **untimeout_func** in the device interrupt routine when you need to cancel a timeout set in the upper-half driver routines.

Return Values

None

NAME

ureadc – copy a character to space described by **uio**(D4) structure

SYNOPSIS

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

int ureadc(int c, uio_t *uiop);
```

Arguments

c	The character to be copied.
uiop	Pointer to the **uio** structure.

DESCRIPTION

ureadc copies the character *c* into the space described by the **uio** structure pointed to by *uiop*.

The **uio_segflg** member of the **uio** structure specifies the type of space to which the copy is made. If **uio_segflg** is set to **UIO_SYSSPACE** the character is copied to a kernel address. If **uio_segflg** is set to **UIO_USERSPACE** the character is copied to a user address.

If the character is successfully copied, **ureadc** updates the appropriate members of the **uio** and **iovec**(D4) structures to reflect the copy (**uio_offset** and **iov_base** are incremented and **uio_resid** and **iov_len** are decremented).

Return Values

ureadc returns **0** on success or an error number on failure.

USAGE

Level

Base only if **uio_segflg** is set to **UIO_USERSPACE**.

Base or Interrupt if **uio_segflg** is set to **UIO_SYSSPACE**.

Synchronization Constraints

May sleep if **uio_segflg** is set to **UIO_USERSPACE**.

Driver-defined basic locks and read/write locks may be held across calls to this function if **uio_segflg** is **UIO_SYSSPACE** but may not be held if **uio_segflg** is **UIO_USERSPACE**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of **uio_segflg**.

Warnings

When holding locks across calls to this function, multithreaded drivers must be careful to avoid creating a deadlock. During the data transfer, page fault resolution might result in another I/O to the same device. For example, this could occur if the driver controls the disk drive used as the swap device.

REFERENCES

iovec(D4), **uio**(D4), **uiomove**(D3), **uwritec**(D3)

NAME

userdma – lock, unlock physical memory in user space

SYNOPSIS

```
#include "sys/types.h"
#include "sys/buf.h"

userdma(void *usr_v_addr, unsigned int num_bytes, int rw);
```

Arguments

usr_v_addr
: User process address space.

num_bytes
: Number of bytes.

rw
: If set to *B_READ*, then the memory space will be readable upon return from this call. If set to *B_WRITE*, the memory will be writable upon return.

DESCRIPTION

userdma prepares memory before a DMA operation into or from a user process' address space. It locks the physical pages associated with *num_bytes* bytes of user virtual memory starting at location *usr_v_addr*. If the rw flag is set to *B_READ*, then the memory space will be readable upon return from this call. If, however, the flag is set to *B_WRITE*, the memory will be writable upon return. **userdma** also invalidates the data cache lines corresponding to the given address and count, if necessary.

When the operation is complete, call **undma** to unlock these pages.

Return Values

If **userdma** is successful, it returns 1; otherwise, it returns 0 and sets the per-process global variable, *u.u_error*, as follows:

[EFAULT]
: The user buffer was outside the allocated address space.

[EAGAIN]
: Total amount of system memory to lock user pages is temporarily insufficient. The **undma** call has no return value.

NAME

uwritec – return a character from space described by **uio**(D4) structure

SYNOPSIS

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

int uwritec(uio_t *uiop);
```

Arguments

uiop	Pointer to the **uio** structure.

DESCRIPTION

uwritec copies a character from the space described by the **uio** structure pointed to by *uiop* and returns the character to the caller.

The **uio_segflg** member of the **uio** structure specifies the type of space from which the copy is made. If **uio_segflg** is set to **UIO_SYSSPACE** the character is copied from a kernel address. If **uio_segflg** is set to **UIO_USERSPACE** the character is copied from a user address.

If the character is successfully copied, **uwritec** updates the appropriate members of the **uio** and **iovec**(D4) structures to reflect the copy (**uio_offset** and **iov_base** are incremented and **uio_resid** and **iov_len** are decremented) and returns the character to the caller.

Return Values

If successful, **uwritec** returns the character. **-1** is returned if the space described by the **uio** structure is empty or there is an error.

USAGE

Level

Base only if **uio_segflg** is set to **UIO_USERSPACE**.

Base or Interrupt if **uio_segflg** is set to **UIO_SYSSPACE**.

Synchronization Constraints

May sleep if **uio_segflg** is set to **UIO_USERSPACE**.

Driver-defined basic locks and read/write locks may be held across calls to this function if **uio_segflg** is **UIO_SYSSPACE** but may not be held if **uio_segflg** is **UIO_USERSPACE**.

Driver-defined sleep locks may be held across calls to this function regardless of the value of **uio_segflg**.

Warnings

When holding locks across calls to this function, multithreaded drivers must be careful to avoid creating a deadlock. During the data transfer, page fault resolution might result in another I/O to the same device. For example, this could occur if the driver controls the disk drive used as the swap device.

REFERENCES

iovec(D4), **uio**(D4), **uiomove**(D3), **ureadc**(D3)

NAME

valusema – return the value associated with a semaphore

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"

valusema (sema_t *semap);
```

DESCRIPTION

valusema returns a snapshot of the semaphore value associated with the semaphore pointed to by *semap*. Because it performs no work, **valusema** is primarily used for assertions.

Because the semaphore value can change immediately after the call, you cannot use **valusema** for conditional semaphore operations. For situations where you need to do this, use **cpsema**(D3X) and **cvsema**(D3X).

To initialize a semaphore, call **initnsema**(D3X) or **initnsema_mutex**(D3X).

Return Values

The returned value of this function is the value of the semaphore pointed to by the *semap* parameter. Of course, if you give this function a bogus semaphore pointer, there is no telling what the function returns.

See Also

ASSERT(D3)

NAME

vme_adapter – determine VME adapter

SYNOPSIS

int vme_adapter(paddr_t *addr***);**

DESCRIPTION

This function takes a VME address and returns the number of the VME adapter to which the address corresponds. This adapter number is required by such functions as **dma_mapalloc**(D3X).

Return Values

If the passed-in address is a valid VME address, **vme_adapter** returns the adapter number; otherwise, it returns –1.

See Also

dma_mapalloc(D3X)

NAME

vme_ivec_alloc – allocate a VME bus interrupt VECTOR

SYNOPSIS

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

int vme_ivec_alloc(uint_t adapter);
```

Arguments

adapter The adapter number identifying which VME bus on the system.

DESCRIPTION

vme_ivec_alloc dynamically allocates an interrupt vector for the specified VME bus. With **vme_ivec_set**(D3X), a driver can allocate and register more than one VME interrupt vector for a single board. **vme_ivec_alloc**(D3X) and **vme_ivec_set**(D3X) are used in **edtinit()** routines.

Return Values

vme_ivec_alloc returns –1 if no vectors remain or the adapter specified is invalid.

See Also

vme_ivec_set(D3X), **vme_ivec_free**(D3X)

NAME

vme_ivec_free – free up a VME bus interrupt VECTOR

SYNOPSIS

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

void vme_ivec_free (int adapter, int vec);
```

Arguments

adapter	The adapter number identifying which VME bus on the system.
vec	The vector allocated from **vme_ivec_alloc**(D3X).

DESCRIPTION

vme_ivec_free returns the specified interrupt vector to the specified VME bus' free list. **vmeivec_free** is called when the driver is unloaded.

Return Values

vme_ivec_free returns –1 if no vector or the adapter specified is invalid.

See Also

vme_ivec_set(D3X), **vme_ivec_alloc**(D3X)

NAME

`vme_ivec_set` – register a VME bus interrupt handler

SYNOPSIS

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

int vme_ivec_set (int adapter, int vec, int (*intr)(int), int arg);
```

Arguments

adapter	The adapter number identifying which VME bus on the system.
vec	The vector allocated from **`vme_ivec_alloc`**(D3X).
intr	A pointer to the driver's interrupt handler.
arg	A value to be passed into the interrupt handler when the interrupt occurs.

DESCRIPTION

`vme_ivec_set` registers the interrupt handler to the specified VME bus interrupt table. With **`vme_ivec_alloc`**(D3X), a driver can allocate and register more than one VME interrupt vector for a single board. **`vme_ivec_alloc`**(D3X) and **`vme_ivec_set`**(D3X) are used in **`edtinit()`** routines.

Return Values

`vme_ivec_set` returns –1 if no vectors remain or the adapter specified is invalid.

See Also

`vme_ivec_alloc`(D3X), **`vme_ivec_free`**(D3X)

NAME

`volatile` – inform the compiler of volatile variables

SYNOPSIS

`volatile`

DESCRIPTION

`volatile` is a compiler directive that causes the variable(s) associated with it not to be affected by the optimizer; that is, memory accesses will be done, in the coded sequence, even if they appear to have no side effects. Pointers to device registers should always be declared volatile.

Return Values

None

NAME

vpsema – perform an atomic "V" and "P" semaphore operation on two semaphores

SYNOPSIS

```
#include "sys/types.h"
#include "sys/param.h"
#include "sys/sema.h"

vpsema(sema_t *sema1p, sema_t *sema2p, int priority);
```

DESCRIPTION

vpsema performs a **vsema** operation with the first semaphore and a **psema** on the second semaphore and the given priority. (See **psema**(D3X) and **vsema**(D3X) for details on these operations.) **vpsema** is atomic in the sense that no other process (on a multiprocessor) can perform a semaphore operation on the first semaphore before the **psema** operation has been performed on the second semaphore. It effectively "swaps" one semaphore for another.

To initialize semaphores, use **initnsema**(D3X) or **initnsema_mutex**(D3X).

Return Values

vpsema returns –1 if a signal interrupts a breakable sleep and *PCATCH* was set; otherwise, it returns 0.

See Also

initnsema(D3X), **initnsema_mutex**(D3X), **psema**(D3X), **vsema**(D3X), **SLEEP_UNLOCK**(D3)

NAME

vsema – perform a "V" or signal semaphore operation

SYNOPSIS

```
#include "sys/types.h"
#include "sys/sema.h"

vsema(sema_t *semap);
```

DESCRIPTION

vsema performs a "V" semaphore operation on the semaphore pointed to by *semap*. The value associated with the semaphore is incremented by 1. If the semaphore value is then less than or equal to 0, a sleeping process is awakened.

vsema expects a pointer to the semaphore as its argument. To allocate or initialize semaphores, use **initnsema**(D3X).

Return Values

vsema returns 0 if no sleeping process was awakened; otherwise, it returns 1.

See Also

initnsema(D3X), **psema**(D3X), **cvsema**(D3X), **SLEEP_UNLOCK**(D3)

NAME

`v_getaddr` – get the user address associated with virtual handle

SYNOPSIS

```
#include "sys/types.h"
#include "sys/immu.h"
#include "sys/region.h"

v_getaddr(vhandl_t *vt);
```

DESCRIPTION

`v_getaddr` gets the user virtual address with which the virtual handle *vt* is associated and writes it to *vt*.

Return Values

None

See Also

`v_gethandle`(D3X), **`v_getlen`**(D3X), **`v_mapphys`**(D3X)

NAME

v_gethandle – get unique identifier associated with virtual handle

SYNOPSIS

```
#include "sys/types.h"
#include "sys/immu.h"
#include "sys/region.h"

v_gethandle(vhandl_t *vt);
```

DESCRIPTION

v_gethandle gets the unique identifier with which the virtual handle *vt* is associated and writes it to *vt*. It is this value only, and not the address of *vt*, which uniquely identifies the virtual handle, *vt*, upon successive calls to driver routines.

Return Values

None

See Also

v_getaddr(D3X), **v_getlen**(D3X), **v_mapphys**(D3X)

NAME

v_getlen – get length of user address space associated with virtual handle

SYNOPSIS

```
#include "sys/types.h"
#include "sys/immu.h"
#include "sys/region.h"

v_getlen (vhandl_t *vt);
```

DESCRIPTION

v_getlen gets the length of the user virtual address space with which the virtual handle *vt* is associated and writes that value to *vt*.

Return Values

None

See Also

v_getaddr(D3X), **v_gethandle**(D3X), **v_mapphys**(D3X)

NAME

v_mapphys – map physical addresses into user address space

SYNOPSIS

```
#include "sys/types.h"
#include "sys/immu.h"
#include "sys/region.h"

v_mapphys(vhandl_ *vt, char *addr, long len);
```

DESCRIPTION

v_mapphys maps *len* bytes of the physical hardware addressed by *addr* into a user's address space. The actual user virtual address space to which *addr* is mapped is determined by *vt*, the "virtual handle" passed to the device driver map routine when the user issues a **mmap**(2) system call.

If *addr* refers to memory and *addr* is either a *kseg0* address or a cached *kseg2* address, *addr* is mapped as a cached address (that is, all loads and stores will access or fill the data cache); otherwise, all references are uncached.

Return Values

In the event of an error, **v_mapphys** returns an errno value; otherwise, it returns 0. The errors include:

[ENOMEM]
Not enough memory was available to allocate page tables for the address space, or the user requested a specific mapping that collides with an address space that cannot be unmapped.

See Also

v_getaddr(D3X), **v_gethandle**(D3X), **v_getlen**(D3X)

NAME

`wakeup` – resume suspended process execution

SYNOPSIS

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

void wakeup(caddr_t event);
```

Arguments

event	Address that was passed to the corresponding call to **`sleep`**(D3) which caused the process to be suspended.

DESCRIPTION

`wakeup` awakens all processes sleeping on the address specified by *event* and makes them eligible for scheduling.

Return Values

None

USAGE

The same *event* argument must be used for corresponding calls to **`sleep`** and **`wakeup`**. It is recommended for code readability and for efficiency to have a one-to-one correspondence between events and **`sleep`** addresses.

Whenever a driver returns from a call to **`sleep`**, it should test to ensure that the event for which the driver slept actually occurred. There is an interval between the time the process that called **`sleep`** is awakened and the time it resumes execution where the state forcing the **`sleep`** may have been reentered. This can occur because all processes waiting for an event are awakened at the same time. The first process selected for execution by the scheduler usually gains control of the event. All other processes awakened should recognize that they cannot continue and should reissue the **`sleep`** call.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

REFERENCES

`sleep`(D3)

NAME

`wbadaddr` – check for bus error when writing to an address

SYNOPSIS

`wbadaddr(char *`*addr***`, int`** *size***`);`**

Arguments

addr	The address of the location to be read.
size	The size in bytes of the location to be read. *size* can be: 1 (one byte), 2 (two bytes = short or half word), or 4 (four bytes = long word).

DESCRIPTION

Call **`wbadaddr`** to determine whether you can write to the specified address location. Typically, you call **`wbadaddr`** from a VME device's **`edtinit()`** function to determine whether a device is still on the present in the system.

Return Values

If the addressed location is writable, **`wbadaddr`** returns 0. Otherwise, **`wbadaddr`** returns 1.

See Also

`badaddr`(D3X)

NAME

WR – get a pointer to the write queue

SYNOPSIS

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

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

Arguments

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

DESCRIPTION

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

Return Values

The pointer to the write queue.

USAGE

Note that when **WR** is passed a write queue pointer as an argument, it returns a pointer to this write queue.

Level

Base or Interrupt.

Synchronization Constraints

Does not sleep.

Driver-defined basic locks, read / write locks, and sleep locks may be held across calls to this function.

Examples

In a STREAMS **open**(D2) routine, the driver or module is passed a pointer to the read queue. The driver or module can store a pointer to a private data structure in the **q_ptr** field of both the read and write queues if it needs to identify the data structures from its **put**(D2) or **srv**(D2) routines.

```
extern struct xxx_dev[];
...
xxxopen(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *crp)
{
   ...
   q->q_ptr = (caddr_t)&xxx_dev[getminor(*devp)];
   WR(q)->q_ptr = (caddr_t)&xxx_dev[getminor(*devp)];
   ...
}
```

REFERENCES

OTHERQ(D3), **RD**(D3)

Kernel Data Structures and Extensions (D4 and D4X)

NAME

intro – introduction to kernel data structures

SYNOPSIS

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

DESCRIPTION

This section describes the kernel data structures a developer might need to use in a device driver.

USAGE

Driver developers should not declare arrays of these structures, as the size of any structure might change between releases. Two exceptions to this are the **iovec**(D4) and **uio**(D4) structures.

Drivers can only reference those structure members described on the manual page. The actual data structures may have additional structure members beyond those described, but drivers must not reference them.

Some structure members are flags fields that consist of a bitmask of flags. Drivers must never directly assign values to these structure members. Drivers should only set and clear flags they are interested in, since the actual implementation may contain unlisted flags.

Data structures that are ‘‘black boxes’’ to drivers are not described in this section. These structures are referenced on the manual pages where they are used. Drivers should not be written to use any of their structure members. Their only valid use is passing pointers to the structures to the particular kernel routines.

NAME

buf – block I/O data transfer structure

SYNOPSIS

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

DESCRIPTION

The **buf** structure is the basic data structure for block I/O transfers.

USAGE

Each block I/O transfer has an associated buffer header. The header contains all the buffer control and status information. For drivers, the buffer header pointer is the sole argument to a block driver **strategy**(D2) routine. Do not depend on the size of the **buf** structure when writing a driver.

It is important to note that a buffer header may be linked in multiple lists simultaneously. Because of this, most of the members in the buffer header cannot be changed by the driver, even when the buffer header is in one of the drivers' work lists.

Buffer headers may be used by the system to describe a portion of the kernel data space for I/O for block drivers. Buffer headers are also used by the system for physical I/O for block drivers. In this case, the buffer describes a portion of user data space that is locked into memory [see **physiock**(D3)].

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

Structure Definitions

The **buf** structure contains the following members:

```
uint_t          b_flags;        /* Buffer status */
struct buf      *b_forw;        /* Kernel/driver list link */
struct buf      *b_back;        /* Kernel/driver list link */
struct buf      *av_forw;       /* Driver work list link */
struct buf      *av_back;       /* Driver work list link */
unsigned int    b_bcount;       /* # of bytes to transfer */
union {
       caddr_t  b_addr;         /* Buffer's virtual address */
} b_un;
daddr_t         b_blkno;        /* Block number on device */
unsigned int    b_resid;        /* # of bytes not transferred */
clock_t         b_start;        /* Request start time */
struct proc     *b_proc;        /* Process structure pointer */
long            b_bufsize;      /* Size of allocated buffer */
```

```
void            (*b_iodone)();  /* Function called by biodone */
void            *b_iochain;     /* link pointer for iodone chain */
dev_t           b_edev;         /* Expanded dev field */
void            *b_private;     /* For driver's use in SVR4MP only*/
```

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

b_flags is a bitmask that stores the buffer status and tells the driver whether to read from or write to the device. To avoid an error condition, the driver must never clear the **b_flags** member or modify its value, except by setting or clearing individual flag bits as described below.

Valid flags are as follows:

B_BUSY	The buffer is in use. The driver may change this flag only if it acquired the buffer with **getrbuf**(D3), and if no I/O operation is in progress.
B_DONE	The data transfer has completed. The driver should not change this flag [see **bioreset**(D3)].
B_ERROR	The driver sets **B_ERROR** to indicate an error occurred during an I/O transfer. On systems where the **bioerror**(D3) function is available, drivers should not access this flag directly.
B_PAGEIO	The buffer is being used in a paged I/O request. If **B_PAGEIO** is set, the **b_pages** field of the buffer header points to a list of page structures sorted by block location on the device. Also, the **b_un.b_addr** field of the buffer header is the offset into the first page of the page list. If **B_PAGEIO** is not set, the **b_pages** field of the buffer header is not used and the **b_un.b_addr** field of the buffer header contains the starting virtual address of the I/O request (in user address space if **B_PHYS** is set or kernel address space otherwise). The driver must not set or clear the **B_PAGEIO** flag.
B_PHYS	The buffer header is being used for physical (direct) I/O to a user data area. The **b_un.b_addr** member contains the starting virtual address of the user data area. Note that **B_PHYS** and **B_PAGEIO** are never set simultaneously and must not be changed by the driver.
B_READ	Data are to be read from the peripheral device into main memory. The driver may change this flag only if it acquired the buffer with **getrbuf**(D3), **geteblk**(D3), or **ngeteblk**(D3), and if no I/O operation is in progress.
B_WRITE	Data are to be transferred from main memory to the peripheral device. **B_WRITE** is a pseudo-flag that occupies the same bit location as **B_READ**. **B_WRITE** cannot be directly tested; it is only detected as the absence of **B_READ** (**!(bp->b_flags&B_READ)**.)

b_forw and **b_back** may only be used by the driver if the buffer was acquired by the driver with the **getrbuf** routine. In that case, these members can be used to link the buffer into driver work lists.

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

b_bcount specifies the number of bytes to be transferred for both paged and non-paged I/O requests. The driver may change this member.

b_un.b_addr is either the virtual address of the I/O request, or an offset into the first page of a page list depending on whether **B_PAGEIO** is set. If it is set, the **b_pages** field of the buffer header points to a sorted list of page structures and **b_un.b_addr** is the offset into the first page. If **B_PAGEIO** is not set, **b_un.b_addr** is the virtual address from which data are read or to which data are written. It represents a user virtual address if **B_PHYS** is set, or a kernel virtual address otherwise. The driver may change this member.

b_blkno identifies which logical block on the device is to be accessed. The driver may have to convert this logical block number to a physical location such as a cylinder, track, and sector of a disk. The driver may change this member only if it allocated the buffer via **geteblk**, **ngeteblk**, or **getrbuf**, and if no I/O operation is in progress.

b_resid indicates the number of bytes not transferred. The driver must set this member prior to calling **biodone**(D3).

b_start holds the time the I/O request was started. It is provided for the driver's use in calculating response time and is set by the driver. Its type, **clock_t**, is an integral type upon which direct integer calculations can be performed. It represents clock ticks.

b_proc contains the process structure address for the process requesting an unbuffered (direct) data transfer to or from a user data area (this member is set to **NULL** when the transfer is buffered). The process table entry is used to perform proper virtual to physical address translation of the **b_un.b_addr** member. The driver should not change this member.

b_bufsize contains the size in bytes of the allocated buffer. The driver may change this member only if it acquired the buffer with **getrbuf**, and if no I/O operation is in progress.

(*b_iodone) identifies a specific driver routine to be called by the system when the I/O is complete. If a routine is specified, the **biodone**(D3) routine does not return the buffer to the system. The driver may change this member if no I/O operation is in progress.

(*b_iochain) If b_iodone has been set by another driver layer, it is important to preserve its value and make sure it is called upon i/o completion. b_iochain is provied for this purpose. For example, a driver that wishes to use b_iodone should save the old value of b_iodone and b_iochain and write the address of these saved values into b_iochain and its completion routines address into b_iodone. When the completion routine is called, it should restore both b_iodone and b_iochain and call biodone() with the buffer again.

b_edev contains the external device number of the device. The driver may change this member only if it allocated the buffer via **geteblk**, **ngeteblk**, or **getrbuf**, and if no I/O operation is in progress.

Warnings

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

DDI/DKI-conforming drivers may only use buffer headers that have been allocated using **geteblk**, **ngeteblk**, or **getrbuf**, or have been passed to the driver **strategy** routine.

REFERENCES

biodone(D3), **bioerror**(D3), **biowait**(D3), **brelse**(D3), **clrbuf**(D3), **freerbuf**(D3), **geteblk**(D3), **geterror**(D3), **getrbuf**(D3), **iovec**(D4), **ngeteblk**(D3), **physiock**(D3), **strategy**(D2), **uio**(D4), **uiophysio**(D3X)

NAME

copyreq – STREAMS transparent **ioctl** copy request structure

SYNOPSIS

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

DESCRIPTION

The **copyreq** structure contains the information necessary to process transparent **ioctl**s.

USAGE

The **copyreq** structure is used in **M_COPYIN** and **M_COPYOUT** messages. The module or driver usually converts an **M_IOCTL** or **M_IOCDATA** message into an **M_COPYIN** or **M_COPYOUT** message. The **copyreq** structure is thus overlaid on top of the **iocblk**(D4) or **copyresp**(D4) structure. The stream head guarantees that the message is large enough to contain the different structures.

Structure Definitions

The **copyreq** structure contains the following members:

```
int      cq_cmd;       /* ioctl command */
cred_t   *cq_cr;       /* user credentials */
uint_t   cq_id;        /* ioctl ID */
caddr_t  cq_addr;      /* copy buffer address */
uint_t   cq_size;      /* number of bytes to copy */
int      cq_flag;      /* for future use */
mblk_t   *cq_private;  /* module private data */
```

The **cq_cmd** field is the **ioctl** command, copied from the **ioc_cmd** field of the **iocblk** structure. If the same message is used, then the **cq_cmd** field directly overlays the **ioc_cmd** field (that is, it need not be copied.)

The **cq_cr** field contains a pointer to the user credentials. It is copied from the **ioc_cr** field of the **iocblk** structure. If the same message is used, then the **cq_cr** field directly overlays the **ioc_cr** field (that is, it need not be copied.)

The **cq_id** field is the **ioctl** ID, copied from the **ioc_id** field of the **iocblk** structure. It is used to uniquely identify the **ioctl** request in the stream. If the same message is used, then the **cq_id** field directly overlays the **ioc_id** field (that is, it need not be copied.)

For an **M_COPYIN** message, the **cq_addr** field contains the user address from which the data are to be copied. For an **M_COPYOUT** message, the **cq_addr** field contains the user address to which the data are to be copied. In both cases, the **cq_size** field contains the number of bytes to copy.

The **cq_flag** field is reserved for future use and should be set to 0 by the module or driver.

The **cq_private** field is a field set aside for use by the driver. It can be used to hold whatever state information is necessary to process the **ioctl**. It is copied to the **cp_private** field in the resultant **M_IOCDATA** message. When the **M_COPYIN** or **M_COPYOUT** message is freed, any message that **cq_private** refers to is not freed by the STREAMS subsystem. It is the responsibility of the module or driver to free it.

REFERENCES

copyresp(D4), **datab**(D4), **iocblk**(D4), **messages**(D5), **msgb**(D4)

NAME

copyresp – STREAMS transparent **ioctl** copy response structure

SYNOPSIS

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

DESCRIPTION

The **copyresp** structure contains information in response to a prior copy request necessary to continue processing transparent **ioctl**s.

USAGE

M_IOCDATA messages, generated by the stream head, contain the **copyresp** structure.

If an **M_IOCDATA** message is reused, any unused fields in the new message should be cleared.

Structure Definitions

The **copyresp** structure contains the following members:

```
int      cp_cmd;       /* ioctl command */
cred_t   *cp_cr;       /* user credentials */
uint_t   cp_id;        /* ioctl ID */
caddr_t  cp_rval;      /* status of request */
mblk_t   *cp_private;  /* module private data */
```

The **cp_cmd** field is the **ioctl** command, copied from the **cq_cmd** field of the **copyreq** structure.

The **cp_cr** field contains a pointer to the user credentials. It is copied from the **cq_cr** field of the **copyreq** structure.

The **cp_id** field is the **ioctl** ID, copied from the **cq_id** field of the **copyreq** structure. It is used to uniquely identify the **ioctl** request in the stream.

The **cq_rval** field contains the return value from the last copy request. If the request succeeded, it is set to 0. Otherwise, if it is non-zero, the request failed. On success, the module or driver should continue processing the **ioctl**. On failure, the module or driver should abort **ioctl** processing and free the message. No **M_IOCNAK** message need be generated.

The **cp_private** field is copied from the **cq_private** field of the **copyreq** structure. It is available so that the module or driver can regain enough state information to continue processing the **ioctl** request. When the **M_IOCDATA** message is freed, any message that **cp_private** refers to is not freed by the STREAMS subsystem. It is the responsibility of the module or driver to free it.

REFERENCES

copyreq(D4), **datab**(D4), **iocblk**(D4), **messages**(D5), **msgb**(D4)

NAME

datab – STREAMS data block structure

SYNOPSIS

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

DESCRIPTION

The **datab** structure describes the data of a STREAMS message.

USAGE

The actual data contained in a STREAMS message is stored in a data buffer pointed to by this structure. A message block structure [**msgb**(D4)] includes a field that points to a **datab** structure.

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

Structure Definitions

The **datab** structure is defined as type **dblk_t** and contains the following members:

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

The **db_base** field points to the beginning of the data buffer. Drivers and modules should not change this field.

The **db_lim** field points to one byte past the end of the data buffer. Drivers and modules should not change this field.

The **db_ref** field contains a count of the number of message blocks sharing the data buffer. If it is greater than 1, drivers and modules should not change the contents of the data buffer. Drivers and modules should not change this field.

The **db_type** field contains the message type associated with the data buffer. This field can be changed by the driver. However, if the **db_ref** field is greater than 1, this field should not be changed.

REFERENCES

free_rtn(D4), **messages**(D5), **msgb**(D4)

NAME

eisa_dma_cb – DMA command block structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/eisa.h>
```

DESCRIPTION

The EISA DMA command block structure is used to control a DMA operation.

USAGE

Each DMA operation requested by a driver is controlled by a command block structure whose fields specify the operation to occur.

The DMA control block specifies the parameters to be programmed for a requestor and a target. The requestor is the hardware device that is requesting the DMA operation, while the target is the target of the operation. The typical case is one in which the requestor is an I/O device and the target is memory.

EISA DMA command block structures should only be allocated via **eisa_dma_get_cb**(D3X). Although drivers may access the structure members listed below, they should not make any assumptions about the size of the structure or the contents of other fields in the structure.

Structure Definitions

The **eisa_dma_cb** structure contains the following members:

```
struct eisa_dma_buf  *reqrbufs;   /* requestor data buffer list */
uchar_t              trans_type;  /* Single/Demand/Block/Cascade */
uchar_t              reqr_path;   /* 8/16/32 */
uchar_t              bufprocess;  /* Single/Chain/Auto-Init */
uchar_t              reqr_bswap;  /* byte swap data on/off */
char                 *procparam;  /* parameter buffer for appl call */
int                  (*proc)();   /* address of application call routines */
```

The following bit-fields defined in the **eisa_dma_cb** structure should be initialized using the appropriate EISA defines described below:

```
cb_cmd;         /* Read/Write/Translate/Verify */
targ_step;      /* Inc/Dec/Hold  */
trans_type;     /* Single/Demand/Block/Cascade */
reqr_path;      /* 8/16/32 */
reqr_timing;    /* A, B, C, ISA compatible */
reqr_ringstop;  /* use channel’s stop registers */
reqr_eop:       /* is EOP input/output */
```

The members of the **eisa_dma_cb** structure are:

reqrbufs is a pointer to a list of DMA buffer structures [see **eisa_dma_buf**(D4)] that describes the requestor of the DMA operation.

bufprocess specifies how the DMA target buffer structures are to be processed. It may have the following values:

EISA_DMA_BUF_SNGL
: Specifies that the target consists of a single DMA Buffer.

EISA_DMA_BUF_CHAIN
: Specifies that the target consists of a chain of DMA Buffers.

reqr_bswap specifies whether data should be byte-swapped between the EISA bus and the host memory bus. It may have one of the following values:

EISA_DMA_BSWAP_ON
: Specifies that byte swapping should be performed.

EISA_DMA_BSWAP_OFF
: Specifies that byte swapping should not be performed.

procparam is the parameter to be passed to the subroutine specified by the **proc** field.

proc specifies the address of a routine to be called when a DMA operation is enabled by **eisa_dma_enable**(D3X). Typically, this is used to program the hardware commands that initiate the DMA operation. The value in the **procparam** field is passed as an argument to this routine. This field may be set to **NULL** if no procedure is to be called.

cb_cmd specifies the command for the DMA operation. It may be one of the following values:

EISA_DMA_CMD_READ
: Specifies a DMA read from the target to the requestor.

EISA_DMA_CMD_WRITE
: Specifies a DMA write from the requestor to the target.

targ_step specifies how the target addresses are to be modified after each transfer. They each may have one of the following values:

EISA_DMA_STEP_INC
: Specifies that the target address is to be incremented following each data transfer.

EISA_DMA_STEP_DEC
: Specifies that the target address is to be decremented following each data transfer

trans_type specifies the transfer type of the operation. It can have one of the following values:

EISA_DMA_TRANS_SNGL
: Specifies that a single transfer is to occur.

EISA_DMA_TRANS_BLCK
Specifies that a block transfer is to occur. This is the only acceptable value for software-initiated transfers.

EISA_DMA_TRANS_DMND
Specifies demand transfer mode, which is a variation on block transfer in which the requestor may provide additional control flow on the transfer.

reqr_path species the size of the data path for the requestor. It may have one of the following values:

EISA_DMA_PATH_8
Specifies that the requestor uses an 8-bit data path.

EISA_DMA_PATH_16
Specifies that the requestor uses a 16-bit data path.

EISA_DMA_PATH_32
Specifies that the requestor uses a 32-bit data path.

reqr_timing specifies the timing mode that requestor hardware uses. Its values can be:

EISA_DMA_TIME_ISA
Specifies that ISA compatibility timing is being used.

EISA_DMA_TIME_A
Specifies that EISA type A timing is being used.

EISA_DMA_TIME_B
Specifies that EISA type B timing is being used.

EISA_DMA_TIME_C
Specifies that EISA type C timing is being used.

reqr_ringstop indicates whether or not the EISA ring buffer feature should be enabled or disabled. It may have one of the following values:

EISA_DMA_RING_OFF
Disable the EISA ring buffer feature.

EISA_DMA_RING_ON
Enable the EISA ring buffer feature.

reqr_eop indicates whether EOP acts as a hardware input or output. Typically EOP operates as an output to generate interrupts. It may have one of the following values:

EISA_DMA_EOP_OUTPUT
EOP is an output.

EISA_DMA_EOP_INPUT
EOP is an input.

REFERENCES

eisa_dma_buf(D4), **eisa_dma_free_cb**(D3X), **eisa_dma_get_cb**(D3X), **eisa_dma_prog**(D3X), **eisa_dma_swstart**(D3X)

NAME

eisa_eisa_dma_buf – EISA DMA buffer descriptor structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/eisa.h>
```

DESCRIPTION

The EISA DMA buffer descriptor structure is used to specify the data to be transferred by a DMA operation.

USAGE

Each DMA operation is controlled by a DMA command block [see **eisa_dma_cb**(D4)] structure that includes a pointer to linked list of **eisa_dma_buf** structures.

Each **eisa_dma_buf** structure provides the physical address and size of a data block involved in a DMA transfer. Scatter/gather operations involving multiple data blocks may be implemented by linking together multiple **eisa_dma_buf**s in a singly-linked list. Each **eisa_dma_buf** includes the virtual address of the next EISA DMA buffer descriptor in the list.

EISA DMA buffer descriptor structures should only be allocated via **eisa_dma_get_buf**(D3X). Although drivers may access the members listed below, they should not make any assumptions about the size of the structure or the contents of other fields in the structure.

Structure Definitions

The **eisa_dma_buf** structure contains the following members:

```
ushort_t             count;      /* size of block*/
paddr_t              address;    /* physical address of data block */
struct eisa_dma_buf  *next_buf;  /* next buffer descriptor */
ushort_t             count_hi;   /* for big blocks */
paddr_t              stopval;    /* ring buffer stop */
```

The members of the **eisa_dma_buf** structure are:

count specifies the low-order 16 bits of the size of the data block in bytes.

address specifies the physical address of the data block.

next_buf specifies the virtual address of the next **eisa_dma_buf** in a linked list of EISA DMA buffer descriptors. It should be **NULL** if the buffer descriptor is the last one in the list. Note that an EISA DMA buffer descriptor allocated by **eisa_dma_get_buf** will be zeroed out initially, thus no explicit initialization is required for this field if a value of **NULL** is desired.

count_hi specifies the high-order 16 bits of the size of the data block in bytes. Since a **eisa_dma_buf** allocated by **eisa_dma_get_buf** is initially zeroed out, no explicit initialization is required for this field if the size of the data block may be specified by a **ushort_t**. **stopval** specifies the physical address used to terminate an EISA ring buffer. This field is used in conjunction with the **reqr_ringstopP field in the eisa_dma_cb structure. It should be left as NULL if this EISA feature is not used.**

REFERENCES

eisa_dma_cb(D4), **eisa_dma_free_buf**(D3X), **eisa_dma_get_buf**(D3X)

NAME

free_rtn – STREAMS driver's message free routine structure

SYNOPSIS

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

DESCRIPTION

A **free_rtn** structure is needed for messages allocated via **esballoc**(D3).

USAGE

Since the driver is providing the memory for the data buffer, a way is needed to notify the driver when the buffer is no longer in use. **esballoc** associates the free routine structure with the message when it is allocated. When **freeb**(D3) is called to free the message and the reference count goes to zero, the driver's message free routine is called, with the argument specified, to free the data buffer.

Structure Definitions

The **free_rtn** structure is defined as type **frtn_t** and contains the following members:

```
void  (*free_func)()  /* driver's free routine */
char  *free_arg       /* argument to free_func() */
```

The **free_func** field specifies the driver's function to be called when the message has been freed. It is called with interrupts from STREAMS devices blocked on the processor on which the function is running.

The **free_arg** field is the only argument to the driver's free routine.

REFERENCES

esballoc(D3), **freeb**(D3)

NAME

iocblk – STREAMS **ioctl** structure

SYNOPSIS

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

DESCRIPTION

The **iocblk** structure describes a user's **ioctl**(2) request.

USAGE

The **iocblk** structure is used in **M_IOCTL**, **M_IOCACK**, and **M_IOCNAK** messages. Modules and drivers usually convert **M_IOCTL** messages into **M_IOCACK** or **M_IOCNAK** messages by changing the type and updating the relevant fields in the **iocblk** structure.

Data cannot be copied to the user's buffer with an **M_IOCACK** message if the **ioctl** is transparent.

No data can be copied to the user's buffer with an **M_IOCNAK** message.

When processing a transparent **ioctl**, the **iocblk** structure is usually overlaid with a **copyreq**(D4) structure. The stream head guarantees that the message is large enough to contain either structure.

Structure Definitions

The **iocblk** structure contains the following members:

```
int     ioc_cmd;    /* ioctl command */
cred_t  *ioc_cr;    /* user credentials */
uint_t  ioc_id;     /* ioctl ID */
uint_t  ioc_count;  /* number of bytes of data */
int     ioc_error;  /* error code for M_IOCACK or M_IOCNAK */
int     ioc_rval;   /* return value for M_IOCACK */
```

The **ioc_cmd** field is the **ioctl** command request specified by the user.

The **ioc_cr** field contains a pointer to the user credentials.

The **ioc_id** field is the **ioctl** ID, used to uniquely identify the **ioctl** request in the stream.

The **ioc_count** field specifies the amount of user data contained in the **M_IOCTL** message. User data will appear in **M_DATA** message blocks linked to the **M_IOCTL** message block. If **ioc_count** is set to the special value **TRANSPARENT**, then the **ioctl** request is ''transparent.'' This means that the user did not use the **I_STR** format of STREAMS **ioctl**s and the module or driver will have to obtain any user data with **M_COPYIN** messages, and change any user data with **M_COPYOUT** messages. In this case, the **M_DATA** message block linked to the **M_IOCTL** message block contains the value of the *arg* parameter in the **ioctl** system call. For an **M_IOCACK** message, the **ioc_count** field specifies the amount of data to copy back to the user's buffer.

The **ioc_error** field can be used to set an error for either an **M_IOCACK** or an **M_IOCNAK** message.

The **ioc_rval** field can be used to set the return value in an **M_IOCACK** message. This will be returned to the user as the return value for the **ioctl** system call that generated the request.

REFERENCES

copyreq(D4), **copyresp**(D4), **datab**(D4), **messages**(D5), **msgb**(D4)

NAME

iovec – data storage structure for I/O using **uio**(D4)

SYNOPSIS

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

DESCRIPTION

The **iovec** structure describes a data storage area for transfer in a **uio** structure. Conceptually, it may be thought of as a base address and length specification.

USAGE

A separate interface does not currently exist for allocating **iovec**(D4) structures when the driver needs to create them itself. Therefore, the driver may either use **kmem_zalloc**(D3) to allocate them, or allocate them statically.

Structure Definitions

The **iovec** structure contains the following members:

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

The driver may only set **iovec** structure members to initialize them for a data transfer for which the driver created the **iovec** structure. The driver must not otherwise change **iovec** structure members. However, drivers may read them. The **iovec** structure members available to the driver are:

iov_base contains the address for a range of memory to or from which data are transferred.

iov_len contains the number of bytes of data to be transferred to or from the range of memory starting at **iov_base**.

REFERENCES

physiock(D3), **uiomove**(D3), **ureadc**(D3), **uwritec**(D3), **uio**(D4)

NAME

linkblk – STREAMS multiplexor link structure

SYNOPSIS

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

DESCRIPTION

The **linkblk** structure contains the information needed by a multiplexing driver to set up or take down a multiplexor link.

USAGE

The **linkblk** structure is embedded in the **M_DATA** portion of the **M_IOCTL** messages generated from the following **ioctl**(2) calls: **I_LINK**, **I_UNLINK**, **I_PLINK**, and **I_PUNLINK** [see **streamio**(7)].

Structure Definitions

The **linkblk** structure contains the following members:

```
queue_t  *l_qtop;  /* lower queue of top stream */
queue_t  *l_qbot;  /* upper queue of bottom stream */
int      l_index;  /* unique ID */
```

The **l_qtop** field is a pointer to the lowest write queue in the upper stream. In other words, it is the write queue of the multiplexing driver. If the link is persistent across closes of the driver, then this field is set to **NULL**.

The **l_qbot** field is a pointer to the upper write queue in the lower stream. The lower stream is the stream being linked under the multiplexor. The topmost read and write queues in the lower stream are given to the multiplexing driver to use for the lower half of its multiplexor processing. The **qinit**(D4) structures associated with these queues are those specified for the lower processing in the multiplexing driver's **streamtab**(D4) structure.

The **l_index** field is a unique ID that identifies the multiplexing link in the system. The driver can use this as a key on which it can multiplex or de-multiplex.

REFERENCES

datab(D4), **iocblk**(D4), **ioctl**(2), **messages**(D5), **msgb**(D4), **qinit**(D4), **streamio**(7), **streamtab**(D4)

NAME

module_info – STREAMS driver and module information structure

SYNOPSIS

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

DESCRIPTION

When a module or driver is declared, several identification and limit values can be set. These values are stored in the **module_info** structure. These values are used to initialize the module's or driver's queues when they are created.

USAGE

After the initial declaration, the **module_info** structure is intended to be read-only. However, the flow control limits (**mi_hiwat** and **mi_lowat**) and the packet size limits (**mi_minpsz** and **mi_maxpsz**) are copied to the **queue**(D4) structure, where they may be modified.

There may be one **module_info** structure per read and write queue, or the driver or module may use the same **module_info** structure for both the read and write queues.

Structure Definitions

The **module_info** structure contains the following members:

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

The **mi_idnum** field is a unique identifier for the driver or module that distinguishes the driver or module from the other drivers and modules in the system.

The **mi_idname** field points to the driver or module name. The constant **FMNAMESZ** limits the length of the name, not including the terminating **NULL**. It is currently set to eight characters.

The **mi_minpsz** field is the default minimum packet size for the driver or module queues. This is an advisory limit specifying the smallest message that can be accepted by the driver or module.

The **mi_maxpsz** field is the default maximum packet size for the driver or module queues. This is an advisory limit specifying the largest message that can be accepted by the driver or module.

The **mi_hiwat** field is the default high water mark for the driver or module queues. This specifies the number of bytes of data contained in messages on the queue such that the queue is considered full and hence flow-controlled.

The **mi_lowat** field is the default low water mark for the driver or module queues. This specifies the number of bytes of data contained in messages on the queue such that the queue is no longer flow-controlled.

REFERENCES

queue(D4)

NAME

msgb – STREAMS message block structure

SYNOPSIS

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

DESCRIPTION

A STREAMS message is made up of one or more message blocks, referenced by a pointer to a **msgb** structure. When a message is on a queue, all fields are read-only to drivers and modules.

USAGE

Structure Definitions

The **msgb** structure is defined as type **mblk_t** and contains the following members:

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

The **b_next** and **b_prev** pointers are used to link messages together on a **queue**(D4). These fields can be used by drivers and modules to create linked lists of messages.

The **b_cont** pointer links message blocks together when a message is composed of more than one block. Drivers and modules can use this field to create complex messages from single message blocks.

The **b_rptr** and **b_wptr** pointers describe the valid data region in the associated data buffer. The **b_rptr** field points to the first unread byte in the buffer and the **b_wptr** field points to the next byte to be written in the buffer.

The **b_datap** field points to the data block [see **datab**(D4)] associated with the message block. This field should never be changed by modules or drivers.

The **b_band** field contains the priority band associated with the message. Normal priority messages and high priority messages have **b_band** set to zero. High priority messages are high priority by virtue of their message type. This field can be used to alter the queuing priority of the message. The higher the priority band, the closer to the head of the queue the message is placed.

The **b_flag** field contains a bitmask of flags that can be set to alter the way the stream head will process the message. Valid flags are:

MSGMARK	The last byte in the message is ‘‘marked.’’ This condition is testable from user level via the **I_ATMARK ioctl**(2).

REFERENCES

allocb(D3), **datab**(D4), **esballoc**(D3), **freeb**(D3), **free_rtn**(D4), **messages**(D5)

NAME

qinit – STREAMS queue initialization structure

SYNOPSIS

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

DESCRIPTION

The **qinit** structure contains pointers to processing procedures and default values for a **queue**(D4). Drivers and modules declare **qinit** structure for their read and write queues, and place the addresses of the structures in their **streamtab**(D4) structure. After the initial declaration, all fields are intended to be read-only.

USAGE

There is usually one **qinit** structure for the read side of a module or driver, and one **qinit** structure for the write side.

Structure Definitions

The **qinit** structure contains the following members:

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

The **qi_putp** field contains the address of the **put**(D2) routine for the **queue**.

The **qi_srvp** field contains the address of the service [**srv**(D2)] routine for the **queue**. If there is no service routine, this field should be set to **NULL**.

The **qi_qopen** field contains the address of the **open**(D2) routine for the driver or module. Only the read-side **qinit** structure need define contain the routine address. The write-side value should be set to **NULL**.

The **qi_qclose** field contains the address of the **close**(D2) routine for the driver or module. Only the read-side **qinit** structure need define contain the routine address. The write-side value should be set to **NULL**.

The **qi_qadmin** field is intended for future use and should be set to **NULL**.

The **qi_minfo** field contains the address of the **module_info**(D4) structure for the driver or module.

The **qi_mstat** field contains the address of the **module_stat** structure for the driver or module. The **module_stat** structure is defined in **/usr/include/sys/strstat.h**. This field should be set to **NULL** if the driver or module does not keep statistics.

REFERENCES

`module_info`(D4), **`queue`**(D4), **`streamtab`**(D4)

NAME

queue – STREAMS queue structure

SYNOPSIS

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

DESCRIPTION

A instance of a STREAMS driver or module consists of two **queue** structures, one for upstream (read-side) processing and one for downstream (write-side) processing.

USAGE

This structure is the major building block of a stream. It contains pointers to the processing procedures, pointers to the next queue in the stream, flow control parameters, and a list of messages to be processed.

Structure Definitions

The **queue** structure is defined as type **queue_t** and contains the following members:

```
struct qinit  *q_qinfo;  /* module or driver entry points */
struct msgb   *q_first;  /* first message in queue */
struct msgb   *q_last;   /* last message in queue */
struct queue  *q_next;   /* next queue in stream */
void          *q_ptr;    /* pointer to private data structure */
ulong_t       q_count;   /* approximate size of message queue */
ulong_t       q_flag;    /* status of queue */
long          q_minpsz;  /* smallest packet accepted by QUEUE */
long          q_maxpsz;  /* largest packet accepted by QUEUE */
ulong_t       q_hiwat;   /* high water mark */
ulong_t       q_lowat;   /* low water mark */
```

The **q_qinfo** field contains a pointer to the **qinit**(D4) structure specifying the processing routines and default values for the queue. This field should not be changed by drivers or modules.

The **q_first** field points to the first message on the queue, or is **NULL** if the queue is empty. This field should not be changed by drivers or modules.

The **q_last** field points to the last message on the queue, or is **NULL** if the queue is empty. This field should not be changed by drivers or modules.

The **q_next** field points to the next queue in the stream. This field should not be changed by drivers or modules.

The **q_ptr** field is a private field for use by drivers and modules. It provides a way to associate the driver's per-minor data structure with the queue.

The **q_count** field contains the number of bytes in messages on the queue in priority band 0. This includes normal messages and high priority messages.

The **q_flag** field contains a bitmask of flags that indicate different queue characteristics. No flags may be set or cleared by drivers or modules. However, the following flags may be tested:

QREADR The queue is the read queue. Absence of this flag implies a write queue.

The **q_minpsz** field is the minimum packet size for the queue. This is an advisory limit specifying the smallest message that can be accepted by the queue. It is initially set to the value specified by the **mi_minpsz** field in the **module_info**(D4) structure. This field can be changed by drivers or modules.

The **q_maxpsz** field is the maximum packet size for the queue. This is an advisory limit specifying the largest message that can be accepted by the queue. It is initially set to the value specified by the **mi_maxpsz** field in the **module_info** structure. This field can be changed by drivers or modules.

The **q_hiwat** field is the high water mark for the queue. This specifies the number of bytes of data contained in messages on the queue such that the queue is considered full, and hence flow-controlled. It is initially set to the value specified by the **mi_hiwat** field in the **module_info** structure. This field can be changed by drivers or modules.

The **q_lowat** field is the low water mark for the queue. This specifies the number of bytes of data contained in messages on the queue such that the queue is no longer flow-controlled. It is initially set to the value specified by the **mi_lowat** field in the **module_info** structure. This field can be changed by drivers or modules.

REFERENCES

getq(D3), **module_info**(D4), **msgb**(D4), **putq**(D3), **qinit**(D4), **strqget**(D3), **strqset**(D3)

NAME

streamtab – STREAMS driver and module declaration structure

SYNOPSIS

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

DESCRIPTION

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

USAGE

Each STREAMS driver or module must have a **streamtab** structure. The **streamtab** structure must be named *prefix***info**, where *prefix* is the driver prefix.

Structure Definitions

The **streamtab** structure contains the following members:

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

The **st_rdinit** field contains a pointer to the read-side **qinit** structure. For a multiplexing driver, this is the **qinit** structure for the upper read side.

The **st_wrinit** field contains a pointer to the write-side **qinit** structure. For a multiplexing driver, this is the **qinit** structure for the upper write side.

The **st_muxrinit** field contains a pointer to the lower read-side **qinit** structure for multiplexing drivers. For modules and non-multiplexing drivers, this field should be set to **NULL**.

The **st_muxwinit** field contains a pointer to the lower write-side **qinit** structure for multiplexing drivers. For modules and non-multiplexing drivers, this field should be set to **NULL**.

REFERENCES

qinit(D4)

NAME

stroptions – stream head option structure

SYNOPSIS

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

DESCRIPTION

The **stroptions** structure, used in an **M_SETOPTS** message, contains options for the stream head.

USAGE

The **M_SETOPTS** message is sent upstream by drivers and modules when they want to change stream head options for their stream.

Structure Definitions

The **stroptions** structure contains the following members:

```
ulong_t   so_flags;    /* options to set */
short     so_readopt;  /* read option */
ushort_t  so_wroff;    /* write offset */
long      so_minpsz;   /* minimum read packet size */
long      so_maxpsz;   /* maximum read packet size */
ulong_t   so_hiwat;    /* read queue high water mark */
ulong_t   so_lowat;    /* read queue low water mark */
uchar_t   so_band;     /* band for water marks */
```

The **so_flags** field determines which options are to be set, and which of the other fields in the structure are used. This field is a bitmask and is comprised of the bit-wise OR of the following flags:

SO_READOPT	Set the read option to that specified by the **so_readopt** field.
SO_WROFF	Set the write offset to that specified by the **so_wroff** field.
SO_MINPSZ	Set the minimum packet size on the stream head read queue to that specified by the **so_minpsz** field.
SO_MAXPSZ	Set the maximum packet size on the stream head read queue to that specified by the **so_maxpsz** field.
SO_HIWAT	Set the high water mark on the stream head read queue to that specified by the **so_hiwat** field.
SO_LOWAT	Set the low water mark on the stream head read queue to that specified by the **so_lowat** field.
SO_ALL	Set all of the above options (**SO_READOPT** | **SO_WROFF** | **SO_MINPSZ** | **SO_MAXPSZ** | **SO_HIWAT** | **SO_LOWAT**).

SO_MREADON	Turn on generation of **M_READ** messages by the stream head.
SO_MREADOFF	Turn off generation of **M_READ** messages by the stream head.
SO_NDELON	Use old TTY semantics for no-delay reads and writes.
SO_NDELOFF	Use STREAMS semantics for no-delay reads and writes.
SO_ISTTY	The stream is acting as a terminal.
SO_ISNTTY	The stream is no longer acting as a terminal.
SO_TOSTOP	Stop processes on background writes to this stream.
SO_TONSTOP	Don't stop processes on background writes to this stream.
SO_BAND	The water marks changes affect the priority band specified by the **so_band** field.

The **so_readopt** field specifies options for the stream head that alter the way it handles **read**(2) calls. This field is a bitmask whose flags are grouped in sets. Within a set, the flags are mutually exclusive. The first set of flags determines how data messages are treated when they are read:

RNORM	Normal (byte stream) mode. **read** returns the lesser of the number of bytes asked for and the number of bytes available. Messages with partially read data are placed back on the head of the stream head read queue. This is the default behavior.
RMSGD	Message discard mode. **read** returns the lesser of the number of bytes asked for and the number of bytes in the first message on the stream head read queue. Messages with partially read data are freed.
RMSGN	Message non-discard mode. **read** returns the lesser of the number of bytes asked for and the number of bytes in the first message on the stream head read queue. Messages with partially read data are placed back on the head of the stream head read queue.

The second set of flags determines how protocol messages (**M_PROTO** and **M_PCPROTO**) are treated during a **read**:

RPROTNORM	Normal mode. **read** fails with the error code **EBADMSG** if there is a protocol message at the front of the stream head read queue. This is the default behavior.
RPROTDIS	Protocol discard mode. **read** discards the **M_PROTO** or **M_PCPROTO** portions of the message and return any **M_DATA** portions that may be present. **M_PASSFP** messages are also freed in this mode.
RPROTDAT	Protocol data mode. **read** treats the **M_PROTO** or **M_PCPROTO** portions of the message as if they were normal data (that is, they are delivered to the user.)

The **so_wroff** field specifies a byte offset to be included in the first message block of every **M_DATA** message created by a **write**(2) and the first **M_DATA** message block created by each call to **putmsg**(2).

The **so_minpsz** field specifies the minimum packet size for the stream head read queue.

The **so_maxpsz** field specifies the maximum packet size for the stream head read queue.

The **so_hiwat** field specifies the high water mark for the stream head read queue.

The **so_lowat** field specifies the low water mark for the stream head read queue.

The **so_band** field specifies the priority band to which the high and / or low water mark changes should be applied.

REFERENCES

datab(D4), **messages**(D5), **msgb**(D4), **read**(2), **streamio**(7)

NAME

uio – scatter / gather I / O request structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/uio.h>
#include <sys/ddi.h>
```

DESCRIPTION

The **uio** structure describes an I / O request that can be broken up into different data storage areas (scatter / gather I / O). A request is a list of **iovec**(D4) structures (base / length pairs) indicating where in user space or kernel space the data are to be read / written.

USAGE

The contents of the **uio** structure passed to the driver through the entry points in section D2 should not be changed directly by the driver. The **uiomove**(D3), **ureadc**(D3), and **uwritec**(D3) functions take care of maintaining the **uio** structure. A block driver may also use the **physiock**(D3) function to perform unbuffered I / O. **physiock** also takes care of maintaining the **uio** structure.

A driver that creates its own **uio** structures for a data transfer is responsible for zeroing it prior to initializing members accessible to the driver. The driver must not change the **uio** structure afterwards; the functions take care of maintaining the **uio** structure.

Note that a separate interface does not currently exist for allocating **uio**(D4) and **iovec**(D4) structures when the driver needs to create them itself. Therefore, the driver may either use **kmem_zalloc**(D3) to allocate them, or allocate them statically.

Structure Definitions

The **uio** structure contains the following members:

```
iovec_t  *uio_iov;    /* Pointer to the start of the iovec */
                      /* array for the uio structure */
int      uio_iovcnt;  /* The number of iovecs in the array */
off_t    uio_offset;  /* Offset into file where data are */
                      /* transferred from or to */
short    uio_segflg;  /* Identifies the type of I/O transfer */
short    uio_fmode;   /* File mode flags */
int      uio_resid;   /* Residual count */
```

The driver may only set **uio** structure members to initialize them for a data transfer for which the driver created the **uio** structure. The driver must not otherwise change **uio** structure members. However, drivers may read them. The **uio** structure members available for the driver to test or set are described below:

uio_iov contains a pointer to the **iovec** array for the **uio** structure. If the driver creates a **uio** structure for a data transfer, an associated **iovec** array must also be created by the driver.

uio_iovcnt contains the number of elements in the **iovec** array for the **uio** structure.

uio_offset contains the starting logical byte address on the device where the data transfer is to occur. Applicability of this field to the driver is device-dependent. It applies to randomly accessed devices, but may not apply to all sequentially accessed devices.

uio_segflg identifies the virtual address space in which the transfer data areas reside. The value **UIO_SYSSPACE** indicates the data areas are within kernel space. The value **UIO_USERSPACE** indicates one data area is within kernel space and the other is within the user space of the current process context.

uio_fmode contains flags describing the file access mode for which the data transfer is to occur. Valid flags are:

FNDELAY	The driver should not wait if the requested data transfer cannot occur immediately; it should terminate the request without indicating an error occurred. The driver's implementation of this flag's implied semantics are subject to device-dependent interpretation.
FNONBLOCK	The driver should not wait if the requested data transfer cannot occur immediately; it should terminate the request, returning the **EAGAIN** error code as the completion status [see **errnos**(D5)]. The driver's implementation of the implied semantics of this flag are subject to device-dependent interpretation.

If the driver creates a **uio** structure for a data transfer, it may set the flags described above in **uio_fmode**.

uio_resid indicates the number of bytes that have not been transferred to or from the data area. If the driver creates a **uio** structure for a data transfer, **uio_resid** is initialized by the driver as the number of bytes to be transferred. Note that a separate interface does not currently exist for allocating

REFERENCES

iovec(D4), **physiock**(D3), **read**(D2), **uiomove**(D3), **ureadc**(D3), **uwritec**(D3), **write**(D2)

Kernel Definitions (D5)

NAME

intro – introduction to kernel **#define**’s

SYNOPSIS

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

DESCRIPTION

This section describes the kernel **#define**’s a developer may need to use in a device driver. Most **#define**’s are specified on the manual page in which they are used. However, some **#define**’s are too general or numerous to include in another manual page. Instead, they have been given a separate page in this section.

USAGE

#include <sys/ddi.h> must always be the last header file included.

NAME

errnos – error numbers

SYNOPSIS

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

DESCRIPTION

The following is a list of the error codes that drivers may return from their entry points, or include in STREAMS messages (for example, **M_ERROR** messages).

EACCES	Permission denied. An attempt was made to access a file in a way forbidden by its file access permissions.
EADDRINUSE	The address requested is already in use.
EADDRNOTAVAIL	The address requested cannot be assigned.
EAFNOSUPPORT	The address family specified is not installed or supported on the host.
EAGAIN	Temporary resource allocation failure; try again later. Drivers can return this error when resource allocation fails, for example, **kmem_alloc**(D3) or **allocb**(D3).
EALREADY	The operation requested is already being performed.
EBUSY	Device is busy. This can be used for devices that require exclusive access.
ECONNABORTED	A received connect request was aborted when the peer closed its endpoint.
ECONNREFUSED	The connection was refused.
ECONNRESET	The connection was reset by the peer entity.
EDESTADDRREQ	The requested operation required a destination address but none was supplied.
EEXIST	Unable to register module for dynamic loading because the module is already statically configured.
EFAULT	Bad address. Drivers should return this error whenever a call to **copyin**(D3) or **copyout**(D3) fails.
EHOSTDOWN	Host is down.
EHOSTUNREACH	No route to host.
EINPROGRESS	The operation requested is now in progress.
EINTR	Interrupted operation. Drivers can return this error whenever an interruptible operation is interrupted by receipt of an asynchronous signal.
EINVAL	Invalid argument. Drivers can return this error for operations that have invalid parameters specified.

EIO	An I/O error has occurred. Drivers can return this error when an input or output request has failed.
EISCONN	The endpoint is already connected.
EMSGSIZE	Message too long. The protocol is such that there is a limit to the size of a message and that limit has been exceeded.
ENETDOWN	The network trying to be reached is down.
ENETRESET	The network dropped the connection because of a reset.
ENETUNREACH	The network trying to be reached is unreachable.
ENOBUFS	No buffer space available.
ENODEV	No such device. Drivers can return this error when an attempt is made to apply an inappropriate function to a device; for example, trying to read a write-only device such as a printer.
ENOMEM	Not enough memory. Drivers can return this error when resource allocation fails and it is either inconvenient or impossible for a retry to occur.
ENOPROTOOPT	The protocol option requested is not available at the level indicated.
ENOSPC	The device is out of free space.
ENOTCONN	The requested operation requires the endpoint to be connected but it is not.
ENXIO	No such device or address. Drivers can return this error when trying to open an invalid minor device, or when trying to perform I/O past the end of a device. This error may also occur when, for example, a tape drive is not online or a disk pack is not loaded on a drive.
EOPNOTSUPP	The operation requested is not supported.
EPERM	Permission denied. Drivers can return this error when a operation is attempted that requires more privilege than the current process has.
EPROTO	Protocol error. Drivers can return this error when they incur a protocol error, such as not being able to generate the proper protocol message because of resource exhaustion, and not being able to recover gracefully.
ETIMEDOUT	The connection timed out.

USAGE

The above examples are not exhaustive.

REFERENCES

geterror(D3)

NAME

messages – STREAMS messages

SYNOPSIS

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

DESCRIPTION

The following is a list of the STREAMS messages types that can be used by drivers and modules.

M_DATA	Data message.
M_PROTO	Protocol control message.
M_BREAK	Control message used to generate a line break.
M_SIG	Control message used to send a signal to processes.
M_DELAY	Control message used to generate a real-time delay.
M_CTL	Control message used between neighboring modules and drivers.
M_IOCTL	Control message used to indicate a user **ioctl**(2) request.
M_SETOPTS	Control message used to set stream head options.
M_IOCACK	High priority control message used to indicate success of an **ioctl** request.
M_IOCNAK	High priority control message used to indicate failure of an **ioctl** request.
M_PCPROTO	High priority protocol control message.
M_PCSIG	High priority control message used to send a signal to processes.
M_READ	High priority control message used to indicate the occurrence of a **read**(2) when there are no data on the stream head read queue.
M_FLUSH	High priority control message used to indicate that queues should be flushed.
M_STOP	High priority control message used to indicate that output should be stopped immediately.
M_START	High priority control message used to indicate that output can be restarted.
M_HANGUP	High priority control message used to indicate that the device has been disconnected.
M_ERROR	High priority control message used to indicate that the stream has incurred a fatal error.
M_COPYIN	High priority control message used during transparent **ioctl** processing to copy data from the user to a STREAMS message.
M_COPYOUT	High priority control message used during transparent **ioctl** processing to copy data from a STREAMS message to the user.

M_IOCDATA	High priority control message used during transparent **ioctl** processing to return the status and data of a previous **M_COPYIN** or **M_COPYOUT** request.
M_PCCTL	High priority control message used between neighboring modules and drivers.
M_PCSETOPTS	High priority control message used to set stream head options.
M_STOPI	High priority control message used to indicate that input should be stopped immediately.
M_STARTI	High priority control message used to indicate that input can be restarted.

REFERENCES

allocb(D3), **copyreq**(D4), **copyresp**(D4), **datab**(D4), **iocblk**(D4), **linkblk**(D4), **msgb**(D4), **put**(D2), **srv**(D2), **stroptions**(D4)

signals(D5)

NAME

signals – signal numbers

SYNOPSIS

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

DESCRIPTION

There are two ways to send a signal to a process. The first, **proc_signal**(D3), can be used by non-STREAMS drivers. The second, by using an **M_SIG** or **M_PCSIG** message, can be used by STREAMS drivers and modules.

The following is a list of the signals that drivers may send to processes.

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.
SIGTSTP	Interactive stop of the process.
SIGURG	Urgent data are available.
SIGWAITING	All LWPs in a process are blocked.
SIGWINCH	The window size has changed.

USAGE

The signal **SIGTSTP** cannot be generated with **proc_signal**. It is only valid when generated from a stream.

REFERENCES

proc_ref(D3), **proc_signal**(D3), **proc_unref**(D3)

Index

A

adjmsg(D3) 50
allocb(D3) 51
ASSERT(D3) 53

B

badaddr(D3X) 54
bcanput(D3) 55
bcopy(D3) 56
biodone(D3) 58
bioerror(D3) 60
biowait(D3) 61
bptophys(D3X) 62
bp_mapin(D3) 63
bp_mapout(D3) 64
brelse(D3) 65
btod(D3X) 66
btop(D3) 67
btopr(D3) 68
buf(D4) 292
bufcall(D3) 69
bzero(D3) 72

C

canput(D3) 73
close(D2) 10
clrbuf(D3) 74
cmn_err(D3) 75
copyb(D3) 78
copyin(D3) 80
copymsg(D3) 82
copyout(D3) 84
copyreq(D4) 296
copyresp(D4) 298
cpsema(D3X) 86
cvsema(D3X) 87

D

datab(D4) 299
datamsg(D3) 88
delay(D3) 90
devflag(D1) 4
dki_dcache_inval(D3X) 91
dki_dcache_wb(D3X) 92
dki_dcache_wbinval(D3X) 93
dma_map(D3X) 94
dma_mapaddr(D3X) 95
dma_mapalloc(D3X) 96
dma_mapfree(D3X) 97

drv_getparm(D3) 98
drv_hztousec(D3) 100
drv_priv(D3) 101
drv_setparm(D3) 102
drv_usectohz(D3) 104
drv_usecwait(D3) 105
dtimeout(D3) 106
dupb(D3) 108
dupmsg(D3) 110

E

eisa_dma_buf(D4X) 304
eisa_dma_cb(D4X) 300
eisa_dma_disable(D3X) 111
eisa_dma_enable(D3X) 112
eisa_dma_free_buf(D3X) 113
eisa_dma_free_cb(D3X) 114
eisa_dma_get_buf(D3X) 115
eisa_dma_get_cb(D3X) 116
eisa_dma_prog(D3X) 117
eisa_dma_stop(D3X) 118
eisa_dma_swstart(D3X) 119
enableok(D3) 120
errnos(D5) 328
esballoc(D3) 121
esbbcall(D3) 123
etoimajor(D3) 124

F

flushband(D3) 125
flushbus(D3X) 126
flushq(D3) 127
freeb(D3) 128
freemsg(D3) 129
freerbuf(D3) 130
freesema(D3X) 131
free_rtn(D4) 306
fubyte(D3X) 132
fuword(D3X) 133

G

geteblk(D3) 134
getemajor(D3) 136
geteminor(D3) 137
geterror(D3) 138
getmajor(D3) 139
getminor(D3) 140
getnextpg(D3X) 141
getq(D3) 142
getrbuf(D3) 143

H

halt(D2) 13
hwcpin(D3X) 145
hwcpout(D3X) 146

I

info(D1) 5
init(D2) 14
initnsema(D3X) 147
initnsema_mutex(D3X) 148
insq(D3) 149
intr(D2) 16
intro(D1) 3

intro(D2) 9
intro(D3) 49
intro(D4) 291
intro(D5) 327
iocblk(D4) 307
ioctl(D2) 18
iovec(D4) 309
itimeout(D3) 151
itoemajor(D3) 153

K

kern_calloc(D3X) 154
kern_free(D3X) 155
kern_malloc(D3X) 156
kmem_alloc(D3) 157
kmem_free(D3) 159
kmem_zalloc(D3) 160
kvtophys(D3X) 162

L

linkb(D3) 163
linkblk(D4) 310
LOCK(D3) 164
LOCK_ALLOC(D3) 166
LOCK_DEALLOC(D3) 168

M

makedevice(D3) 169
map(D2X) 20
max(D3) 171
messages(D5) 330
min(D3) 172
mmap(D2) 22
module_info(D4) 311
msgb(D4) 313
msgdsize(D3) 173
msgpullup(D3) 174

N

ngeteblk(D3) 175
noenable(D3) 177

O

open(D2) 24
OTHERQ(D3) 178

P

pcmsg(D3) 180
phalloc(D3) 182
phfree(D3) 183
physiock(D3) 184
pio_andb_rmw(D3X) 186
pio_andh_rmw(D3X) 187
pio_andw_rmw(D3X) 188
pio_badaddr(D3X) 189
pio_bcopyin(D3X) 190
pio_bcopyout(D3X) 191
pio_mapaddr(D3X) 192
pio_mapalloc(D3X) 193
pio_mapfree(D3X) 194
pio_orb_rmw(D3X) 195
pio_orh_rmw(D3X) 196

pio_orw_rmw(D3X) 197
pio_wbadaddr(D3X) 198
poll(D2) 28
pollwakeup(D3) 199
pptophys(D3X) 200
prefix(D1) 6
print(D2) 30
proc_ref(D3) 201
proc_signal(D3) 202
proc_unref(D3) 203
psema(D3X) 204
ptob(D3) 205
put(D2) 31
putbq(D3) 206
putctl(D3) 207
putctl1(D3) 209
putnext(D3) 210
putq(D3) 211

Q

qenable(D3) 212
qinit(D4) 315
qreply(D3) 213
qsize(D3) 214
queue(D4) 317

R

RD(D3) 215
read(D2) 34
rmalloc(D3) 216
rmallocmap(D3) 219
rmalloc_wait(D3) 220
rmfree(D3) 221
rmfreemap(D3) 223
rmvb(D3) 224
rmvq(D3) 226

S

SAMESTR(D3) 227
scsi_alloc(D3X) 228
scsi_command(D3X) 229
scsi_free(D3X) 230
scsi_info(D3X) 231
sgset(D3X) 232
signals(D5) 332
size(D2) 35
sleep(D3) 233
SLEEP_ALLOC(D3) 235
SLEEP_DEALLOC(D3) 236
SLEEP_LOCK(D3) 237
SLEEP_LOCKAVAIL(D3) 238
SLEEP_LOCK_SIG(D3) 239
SLEEP_TRYLOCK(D3) 241
SLEEP_UNLOCK(D3) 242
spl(D3) 243
srv(D2) 36
start(D2) 39
strategy(D2) 40
strcat(D3) 245
strcpy(D3) 246
streams_interrupt(D3X) 247
STREAMS_TIMEOUT(D3X) 248
streamtab(D4) 319
strlog(D3) 249
stroptions(D4) 320
strqget(D3) 251

strqset(D3) 253
subyte(D3X) 254
suword(D3X) 255

T

TRYLOCK(D3) 256

U

uio(D4) 323
uiomove(D3) 257
uiophysio(D3X) 259
unbufcall(D3) 261
undma(D3X) 263
unlinkb(D3) 264
unload(D2) 42
UNLOCK(D3) 265
unmap(D2X) 43
untimeout(D3) 266
untimeout_func(D3X) 267
ureadc(D3) 268
userdma(D3X) 270
uwritec(D3) 271

V

valusema(D3X) 273
vme_adapter(D3X) 274
vme_ivec_alloc(D3X) 275
vme_ivec_free(D3X) 276
vme_ivec_set(D3X) 277
volatile(D3X) 278
vpsema(D3X) 279
vsema(D3X) 280
v_getaddr(D3X) 281
v_gethandle(D3X) 282
v_getlen(D3X) 283
v_mapphys(D3X) 284

W

wakeup(D3) 285
wbadaddr(D3X) 286
WR(D3) 287
write(D2) 44

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2183-003.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389