UNIX® SYSTEM V RELEASE 4

INCLUDES MULTIPROCESSING

DEVICE DRIVER INTERFACE / DRIVER-KERNEL INTERFACE REFERENCE MANUAL

Intel Processors

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy and completeness of all information in this document, USL assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors, omissions, or statements result from negligence, accident, or any other cause. USL further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. **USL disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose**. USL makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of any license to make, use or sell equipment constructed in accordance with this description.

USL reserves the right to make changes to any products herein without further notice.

TRADEMARKS

UNIX is a registered trademark of UNIX System Laboratories, Inc. in the USA and other countries. Intel386 and Intel486 are trademarks of Intel Corp.

10 9 8 7 6 5 4 3 2

ISBN 0-13-879529-0

UNIX
PRESS
A Prentice Hall Title

Most of the routines, functions, and structures described in this manual are part of both the DDI and the DKI (cross-referenced by D*x*DK). As Figure 1 shows, drivers written to conform to both interfaces are portable to all computers supporting UNIX System V Release 4 Multi-Processor for Intel Processors, and they will be compatible through and beyond Release 4 Multi-Processor.

NOTE

Note that drivers written to conform with this version of the DDI/DKI may not run on systems running UNIX System V Release 4 or Release 4.1 Enhanced Security, as those releases do not implement the new multiprocessor interfaces.

However, a driver written to conform to both interfaces is not guaranteed to be *binary* compatible with future releases of the operating system. Binary compatibility requires more than just interface definition. It also requires that values for **`#define`**'s be standardized, for example. The DDI/DKI is a source code interface. Following it is a necessary, but not sufficient, condition for binary compatibility. To understand more completely what is meant by "portable" and "compatible" for the DDI and DKI, the scope of each interface must be more thoroughly explained.

Porting

Software is usually considered portable if it can be adapted to run in a different environment at a lower cost than if one were to rewrite it. The new environment may include a different processor, operating system, and even the language in which the program is written, if a language translator is available. More often, however, software is ported between environments that share an operating system, processor, and source language. The source code is modified to accommodate the differences in compilers, processors, or releases of the operating system.

In the past, device drivers did not port easily for one or more of the following reasons:

- To enhance functionality, members had been added to kernel data structures accessed by drivers, or the sizes of existing members had been redefined.

- Driver–Hardware. Most hardware drivers include an interrupt handling entry point, and may also perform direct memory access (DMA). These and other hardware-specific interactions make up the driver/hardware interface.
- Driver–Boot/Configuration Software. At boot time, the existence of a driver is made known to the system through information in system files, enabling the system to include the driver. The interaction between the driver and the boot and configuration software is the third interface affecting drivers. Refer to the sections on Installable Drivers (ID) in Chapter 3 of the *Integrated Software Development Guide* for more information on this.

Scope of the Device Driver Interface (DDI)

The primary goal of DDI is to facilitate both source and binary portability across successive releases of UNIX System V on a particular machine. Implicit in this goal is an important fact. Although there is only one DKI, each processor product has its own DDI. Therefore, if a driver is ever to be ported to different hardware, special attention must be paid to the machine-specific routines that make up the "DDI only" part of a driver. These include, but are not confined to, the driver/hardware interface (as described in the previous section). Some processor-specific functionality also may belong to the driver/kernel interface, and may not be easy to locate.

To achieve the goal of source and binary compatibility, the functions, routines, and structures specified in the DDI must be used according to these rules.

- Drivers cannot access system state structures (for example, **u** and **sysinfo**) directly.
- For structures external to the driver that may be accessed directly, only the utility functions provided in Section 3 of this manual should be used. More generally, these functions should be used wherever possible.
- The header file **ddi.h** must be included at the end of the list of system header files. This header file "undefines" several macros that are reimplemented as functions. Device driver-specific include files should be listed after **ddi.h** to insure only the DDI/DKI interface is used by the driver.

- Single-threaded drivers which conform to the DDI/DKI will be portable across uniprocessor implementations which support the DDI/DKI.
- Multiprocessor implementations which support the DDI/DKI are not required to support single-threaded drivers that conform to the DDI/DKI, although some multiprocessor implementations may choose to support such drivers by preventing concurrent execution of code within a given single-threaded driver.

Driver writers are encouraged to write multithreaded rather than single-threaded drivers, as these will be more widely portable and can benefit from the parallelism inherent on a multiprocessor. Writing a multithreaded driver requires that shared data within the driver be protected against certain forms of concurrent access. This is done by using appropriate locking primitives to prevent concurrent execution of code which accesses a given piece of shared data. This document defines interfaces to several types of locking and synchronization primitives, namely basic locks, read/write locks, sleep locks and synchronization variables. Basic locks and read/write locks are intended for use within multithreaded drivers, while sleep locks and synchronization variables are useful in both single-threaded and multithreaded drivers. The characteristics of the various locking and synchronization primitives are described on the relevant manual pages in Section 3.

Audience

This manual is for experienced C programmers responsible for creating, modifying, or maintaining drivers that run on UNIX System V Release 4 Multi-Processor for Intel Processors and beyond. It assumes that the reader is familiar with UNIX system internals and the advanced capabilities of the C Programming Language. In addition, programmers writing multithreaded drivers are assumed to be familiar with the fundamentals of concurrent programming and the appropriate use of locking primitives to protect shared data.

The manual contains five sections:

D1	driver data definitions
D2	driver entry points
D3	kernel functions used by drivers
D4	kernel data structures accessed by drivers
D5	kernel **#define**'s used by drivers

Each section number is suffixed with a letter indicating the interfaces covered. The suffixes used are:

D	Device Driver Interface only (DDI)
K	Driver–Kernel Interface only (DKI)
DK	both DDI and DKI
X	DDI-only Platform-specific Interface

For example, **open**(D2DK) refers to the **open** entry point routine for a driver, not to the **open**(2) system call documented in the *Programmer's Reference Manual*. For clarity, the platform-specific manual pages have been put in an appendix, separate from the rest of the DDI/DKI manual pages.

Reference pages contain the following headings, where applicable:

- NAME gives the routine's name and a short summary of its purpose.
- SYNOPSIS summarizes the routine's calling and return syntax.
- ARGUMENTS describes each of the routine's arguments.
- DESCRIPTION provides general information about the routine.
- STRUCTURE MEMBERS describes all accessible data structure members.
- RETURN VALUE summarizes the return value from the function.
- LEVEL gives an indication of when the routine can be used.
- NOTES provides restrictions on use and cautionary information.
- SEE ALSO gives sources for further information.
- EXAMPLE provides an example of common usage.

STREAMS

The *Programmer's Guide: STREAMS* tells how to write drivers and access devices that use the STREAMS driver interface for character access.

The *Programmer's Guide: Networking Interfaces* provides detailed information, with examples, on the Section 3N library that comprises the UNIX system Transport Level Interface (TLI).

The *Programmer's Guide: ANSI C and Programming Support Tools* includes instructions on using a number of UNIX utilities, including **make** and SCCS.

Operating Systems

The UNIX System V reference manuals are the standard reference materials for the UNIX operating system. This information is organized into three manuals, published separately for each system:

- The *User's Reference Manual/System Administrator's Reference Manual* includes information on UNIX system user-level commands (Section 1) and administrative commands (Section 1M).
- The *Programmer's Reference Manual: Operating System API* includes information on UNIX system calls (Section 2) and C language library routines (Section 3).
- The *System Files and Devices Reference Manual* includes information on UNIX system file formats (Section 4), miscellaneous facilities (Section 5), and special device files (Section 7).

D4. Data Structures

D5. Kernel Defines

Appendix A: Migration from Release 3.2 to Release 4 Multi-Processor

Appendix B: Migration from Release 4 to Release 4 Multi-Processor

NAME

devflag – driver flags

SYNOPSIS

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

int prefixdevflag = 0;
```

DESCRIPTION

Every driver must define a global integer containing a bitmask of flags that indicate its characteristics to the system. The valid flags that may be set are:

D_DMA The driver does DMA (Direct Memory Access).

D_TAPE The driver controls a tape device (mount read-only).

D_NOBRKUP The driver understands the **B_PAGEIO** flag in the buffer header (the I/O job is not broken up along page boundaries into multiple jobs by the kernel).

D_MP The driver is multithreaded (it handles its own locking and serialization).

If no flags are set for the driver, then *prefix***devflag** should be set to 0.

SEE ALSO

Integrated Software Development Guide

NAME

prefix – driver prefix

SYNOPSIS

```
int prefixclose();
int prefixopen();
  ...
```

DESCRIPTION

Every driver must define a unique prefix, whose maximum length is implementation-defined. The prefix is usually specified in a configuration file. Driver entry points names are created by concatenating the driver prefix with the name for the entry point. This enables driver entry points to be identified by configuration software and decreases the possibility of global symbol collisions in the kernel.

SEE ALSO

devflag(D1D), **info**(D1D), **chpoll**(D2DK), **close**(D2DK), **halt**(D2D), **init**(D2D), **intr**(D2D), **ioctl**(D2DK), **mmap**(D2DK), **open**(D2DK), **print**(D2DK), **put**(D2DK), **read**(D2DK), **size**(D2DK), **srv**(D2DK), **start**(D2DK), **strategy**(D2DK), **write**(D2DK)

EXAMPLE

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**.

NAME

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

SYNOPSIS

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

int prefixchpoll(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. Valid events are:

POLLIN	Data are 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 are available to be read.
POLLWRNORM	Normal data may be written without blocking (same as **POLLOUT**).
POLLRDBAND	Out-of-band data are available to be read.
POLLWRBAND	Out-of-band data may be written without blocking.

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 **chpoll** 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)].

A driver that supports polling must provide a **pollhead** structure for each minor device supported by the driver. The driver must use **phalloc**(D3DK) to allocate the **pollhead** structure. It can be freed later, if necessary, with **phfree**(D3DK). 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. Drivers should not depend on the size of the **pollhead** structure.

The driver must implement the polling discipline itself. Each time the driver detects a pollable event, it should call **pollwakeup**(D3DK), 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.

NAME

close – relinquish access to a device

SYNOPSIS [Block and Character]

```
#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 prefixclose(dev_t dev, int flag, int otyp, cred_t *crp);
```

ARGUMENTS

dev Device number.

flag File status flag. Possible flag values and their definitions can be found in **open**(D2DK).

otyp Parameter supplied so that the driver can determine how many times a device was opened and for what reasons. The values are mutually exclusive.

- **OTYP_BLK** Close was through 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.

crp Pointer to the user credential structure.

SYNOPSIS [STREAMS]

```
#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 prefixclose(queue_t *q, int flag, cred_t *crp);
```

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

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.

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.

RETURN VALUE

The **close** routine should return 0 for success, or the appropriate error number. Refer to **errnos**(D5DK) 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.

SEE ALSO

open(D2DK), **drv_priv**(D3DK), **qprocsoff**(D3DK), **unbufcall**(D3DK), **untimeout**(D3DK), **queue**(D4DK), **errnos**(D5DK)

NAME

init – initialize a device

SYNOPSIS

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

DESCRIPTION

init and **start**(D2DK) routines are used to initialize drivers and the devices they control. **init** routines are executed during system initialization, and can be used in drivers that do not require system services to be initialized. **start** routines are executed after system services are enabled.

init routines can perform functions such as:

allocating memory for private buffering schemes

mapping a device into virtual address space

initializing hardware (for example, system generation or resetting the board)

Functions that may result in the caller sleeping, or that require user context, such as **SV_WAIT**(D3DK), may not be called. Any function that provides a flag to prevent it from sleeping must be called such that the function does not sleep. Also, **init** routines are executed before interrupts are enabled.

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

ASSERT	drv_usectohz	physmap
bcopy	drv_usecwait	physmap_free
btop/btopr	etoimajor	repinsb/repinsl/ repinsw
bzero	getemajor	repoutsb/repoutsl/ repoutsw
cmn_err	geteminor	rmalloc
dma_disable	getmajor	rmallocmap
dma_enable	getminor	rmfreemap
dma_free_buf	inb/inl/inw	rmfree
dma_free_cb	itoemajor	RWLOCK_ALLOC
dma_get_best_mode	kmem_alloc	SLEEP_ALLOC
dma_get_buf	kmem_free	SV_ALLOC
dma_get_cb	kmem_zalloc	vtop
dma_prog	LOCK_ALLOC	
dma_stop	makedevice	
dma_swsetup	max/min	
dma_swstart	outb/outl/outw	
drv_getparm	phalloc	
drv_hztousec	phfree	

NOTES

This entry point is optional.

RETURN VALUE

None.

SEE ALSO

start(D2DK)

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**(D3DK) or **SV_WAIT**(D3DK) to await the completion of an operation, the **intr** routine must call **biodone**(D3DK) or **SV_SIGNAL**(D3DK) 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**(D3D) for more information.

NOTES

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

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)

uiomove(D3DK), **ureadc**(D3DK), and **uwritec**(D3DK) cannot be used in an interrupt routine when the **uio_segflg** member of the **uio**(D4DK) structure is set to **UIO_USERSPACE** (indicating a transfer between user and kernel space).

RETURN VALUE

None.

SEE ALSO

biodone(D3DK), **spl**(D3D), **SV_SIGNAL**(D3DK)

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)
```

RETURN VALUE

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.

SEE ALSO

open(D2DK), **copyin**(D3DK), **copyout**(D3DK), **drv_priv**(D3DK), **errnos**(D5DK)

SEE ALSO

`kvtoppid`(D3DK), `phystoppid`(D3D)

`mmap`(2)

ARGUMENTS [STREAMS]

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. Valid values are the same as those listed above.

sflag STREAMS flag. Values are mutually exclusive and are given as follows:

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.

crp Pointer to the user credential structure.

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 STREAMS module **open** routine is called by the kernel during an **I_PUSH ioctl**(2) or an autopush-style open [see **autopush**(1M)].

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

enable 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

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**(D3DK)].

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

NAME

print – display a driver message on the system console

SYNOPSIS

```
#include <sys/types.h>
#include <sys/errno.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. To display the message on the console, the driver should use the **cmn_err**(D3DK) function.

NOTES

This entry point is optional.

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

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

RETURN VALUE

Ignored.

SEE ALSO

cmn_err(D3DK)

```
        }
        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);
```

NAME

read – read data from a device

SYNOPSIS

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

int prefixread(dev_t dev, uio_t *uiop, cred_t *crp);
```

ARGUMENTS

dev Device number.

uiop Pointer to the **uio**(D4DK) 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. 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**(D3DK) function provides a convenient way to copy data using the **uio** structure.

Block drivers that provide a character interface can use **physiock**(D3DK) to perform the data transfer with the driver's **strategy**(D2DK) routine.

NOTES

This interface is optional.

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

RETURN VALUE

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

SEE ALSO

strategy(D2DK), **write**(D2DK), **drv_priv**(D3DK), **physiock**(D3DK), **uiomove**(D3DK), **ureadc**(D3DK), **uio**(D4DK), **errnos**(D5DK)

NAME

srv – service queued messages

SYNOPSIS

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

ARGUMENTS

q Pointer to the queue.

DESCRIPTION

The 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. Service routines also provide a way to recover from resource allocation failures.

A message is first passed to a module's or driver's **put**(D2DK) 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**(D4DK)] has limit values it uses to implement flow control. Tunable 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 process all messages on its queue, 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. The framework for the canonical service procedure algorithm is as follows:

put routines can interrupt and run concurrently with service routines.

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

Drivers and modules should not call service routines directly. **qenable**(D3DK) should be used to schedule service routines to run.

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.

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

RETURN VALUE

Ignored.

SEE ALSO

STREAMS Programmer's Guide
put(D2DK), **bcanputnext**(D3DK), **bufcall**(D3DK), **canputnext**(D3DK), **getq**(D3DK), **pcmsg**(D3DK), **putbq**(D3DK), **putnext**(D3DK), **putq**(D3DK), **qenable**(D3DK), **timeout**(D3DK), **datab**(D4DK), **msgb**(D4DK), **qinit**(D4DK), **queue**(D4DK)

NAME

strategy – perform block I/O

SYNOPSIS

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

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

ARGUMENTS

bp Pointer to the buffer header.

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 kernel function **physiock**(D3DK)), to support the raw character interface of a block device from **read**(D2DK), **write**(D2DK) or **ioctl**(D2DK). The **strategy** routine's responsibility is to set up and initiate the data transfer.

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, **bioerror**(D3DK) should be used to set the buffer error number to **ENXIO**, the buffer should be marked done by calling **biodone**(D3DK), 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 will 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**(D3DK) 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 in **bp->b_flags**. Here, the starting 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**. These will not occur unless the driver has set the **D_NOBRKUP** flag [see **devflag**(D1D)]. 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**(D3DK) to get a virtually contiguous mapping for the pages. If the driver wants to use physical addresses, it can also use **bp_mapin**, but only transfer one page at a time. The physical address can be obtained by calling **vtop**(D3D) for each page in the virtual range. The size of a page can be determined by calling **ptob**(D3DK). However, a more efficient way to use physical addresses is to use **getnextpg**(D3DK) and **pptophys**(D3D) for each page in the page list.

NAME

write – write data to a device

SYNOPSIS

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

int prefixwrite(dev_t dev, uio_t *uiop, cred_t *crp);
```

ARGUMENTS

dev Device number.

uiop Pointer to the **uio**(D4DK) 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. 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**(D3DK) function provides a convenient way to copy data using the **uio** structure.

Block drivers that provide a character interface can use **physiock**(D3DK) to perform the data transfer with the driver's **strategy**(D2DK) routine.

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 asynchronous to the user's request, losing the ability to return an error with the associated request.

NOTES

This interface is optional.

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

RETURN VALUE

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

SEE ALSO

read(D2DK), **strategy**(D2DK), **drv_priv**(D3DK), **physiock**(D3DK), **uiomove**(D3DK), **uwritec**(D3DK), **uio**(D4DK), **errnos**(D5DK)

NAME

adjmsg – trim bytes from a message

SYNOPSIS

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

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

ARGUMENTS

mp Pointer to the message to be trimmed.

len The number of bytes to be removed.

DESCRIPTION

adjmsg removes bytes from a message. | *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.

RETURN VALUE

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

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.

SEE ALSO

msgb(D4DK)

SEE ALSO

Programmer's Guide: STREAMS

bufcall(D3DK), **esballoc**(D3DK), **esbbcall**(D3DK), **freeb**(D3DK), **msgb**(D4DK)

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 7). Otherwise, the message type is set to **M_ERROR** (line 8). Line 9 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 10). If it is not set, *q* points to a write queue, and on line 11 the **RD**(D3DK) function is used to find the corresponding read queue. In line 12, the **putnext**(D3DK) 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     if ((bp = allocb(1, BPRI_HI)) == NULL)
 7           return(0);
 8     bp->b_datap->db_type = M_ERROR;
 9     *bp->b_wptr++ = err;
10     if (!(q->q_flag & QREADR))
11           q = RD(q);
12     putnext(q, bp);
13     return(1);
14  }
```

NAME

bcanput – test for flow control in specified priority band

SYNOPSIS

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

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

ARGUMENTS

q Pointer to the message queue.

pri Message priority.

DESCRIPTION

bcanput tests if there is room for a message in priority band *pri* of the queue pointed to by *q*. The queue *must* have a service procedure.

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

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.

RETURN VALUE

bcanput returns 1 if a message of priority *pri* can be placed on the queue. 0 is returned if a message of priority *pri* cannot be enqueued because of flow control within the priority band.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

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

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The *q* argument may not reference **q_next** (for example, an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI). **bcanputnext(q)** is provided as a multiprocessor-safe equivalent to the common call **bcanput(q->q_next)**, which is no longer allowed [see **bcanputnext**(D3DK)].

SEE ALSO

bcanputnext(D3DK), **canput**(D3DK), **canputnext**(D3DK), **putbq**(D3DK), **putnext**(D3DK)

SEE ALSO

bcanput(D3DK), **canput**(D3DK), **canputnext**(D3DK), **putbq**(D3DK), **putnext**(D3DK)

```
#define RAMDNBLK    1000          /* number of blocks in the 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);
}
```

```
1  #define RAMDNBLK    1000          /* Number of blocks in RAM disk */
2  #define RAMDBSIZ     512          /* Number of bytes per block */
3  char ramdblks[RAMDNBLK][RAMDBSIZ]; /* Array containing RAM disk */

4  ramdstrategy(bp)
5   register struct buf *bp;
6  {
7   register 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  }
     . . .
```

NAME

biowait – suspend processes pending completion of block I/O

SYNOPSIS

```
#include <sys/types.h>
#include <sys/buf.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. Block drivers that have allocated their own buffers with **geteblk**(D3DK), **getrbuf**(D3DK), or **ngeteblk**(D3DK) 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**(D3DK) in their I/O completion handlers to signal **biowait** when the I/O transfer is complete.

RETURN VALUE

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

LEVEL

Base Only.

NOTES

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.

SEE ALSO

intr(D2D), **strategy**(D2DK), **biodone**(D3DK), **geteblk**(D3DK), **getrbuf**(D3DK), **ngeteblk**(D3DK), **buf**(D4DK)

NAME

bp_mapout – deallocate virtual address space for buffer page list

SYNOPSIS

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

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

ARGUMENTS

bp Pointer to the buffer header structure.

DESCRIPTION

This function deallocates the system virtual address space associated with a buffer header page list. The virtual address space must have been allocated by a previous call to **bp_mapin**(D3DK). Drivers should not reference any virtual addresses in the mapped range after **bp_mapout** has been called.

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

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

SEE ALSO

bp_mapin(D3DK), **buf**(D4DK)

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.

For example, if the page size is 2048, then **btop(4096)** and **btop(4097)** both return 2, and **btop(4095)** returns 1.

btop(0) returns 0.

RETURN VALUE

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

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

btopr(D3DK), **ptob**(D3DK)

NAME

bufcall – call a function when a buffer becomes available

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.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**(D3DK) request).

pri Priority of the **allocb** allocation request (**BPRI_LO**, **BPRI_MED**, or **BPRI_HI**).

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

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

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

RETURN VALUE

If successful, **bufcall** returns a non-zero value that identifies the scheduling request. This non-zero identifier may be passed to **unbufcall**(D3DK) to cancel the request. If any failure occurs, **bufcall** returns 0.

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.

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**.

SEE ALSO

allocb(D3DK), **esballoc**(D3DK), **esbbcall**(D3DK), **itimeout**(D3DK), **unbufcall**(D3DK)

EXAMPLE

The purpose of this service routine [see **srv**(D2DK)] 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 21), 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 23) and return. The STREAMS flow control mechanism will re-enable us later when messages can be sent. If **canputnext**(D3DK) succeeded, we try to allocate

```
34                      modp->m_type = TIMEOUT;
35                  } else {
36                      modp->m_type = BUFCALL;
37                  }
38                  UNLOCK(modp->m_lock, pl);
39                  return;
40              }
41              hp = (struct hdr *)bp->b_wptr;
42              hp->h_size = msgdsize(mp);
43              hp->h_version = 1;
44              bp->b_wptr += sizeof(struct hdr);
45              bp->b_datap->db_type = M_PROTO;
46              bp->b_cont = mp;
47              putnext(q, bp);
48      }
49  }
50  modcall(q)
51      queue_t *q;
52  {
53      struct mod *modp;
54      pl_t pl;
55      modp = (struct mod *)q->q_ptr;
56      pl = LOCK(modp->m_lock, plstr);
57      modp->m_type = 0;
58      UNLOCK(modp->m_lock, pl);
59      qenable(q);
60  }
```

NAME

canput – test for room in a message queue

SYNOPSIS

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

int canput(queue_t *q);
```

ARGUMENTS

q Pointer to the message queue.

DESCRIPTION

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

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.

RETURN VALUE

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.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

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

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The *q* argument may not reference **q_next** (for example, an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI). **canputnext(q)** is provided as a multiprocessor-safe equivalent to the common call **canput(q->q_next)**, which is no longer allowed [see **canputnext**(D3DK)].

SEE ALSO

bcanput(D3DK), **bcanputnext**(D3DK), **canputnext**(D3DK), **putbq**(D3DK), **putnext**(D3DK)

NAME

clrbuf – erase the contents of a buffer

SYNOPSIS

```
#include <sys/types.h>
#include <sys/buf.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**(D4DK) 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.

If the buffer has the **B_PAGEIO** 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**(D3DK).

RETURN VALUE

None.

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

bp_mapin(D3DK), **buf**(D4DK)

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.

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.

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

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.

SEE ALSO

print(D2DK)

crash(1M) in the *System Administrator's Reference Manual*

printf(3S) in the *Programmer's Reference Manual*

EXAMPLE

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      register 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 */
```

NAME

copyb – copy a message block

SYNOPSIS

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

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

ARGUMENTS

bp Pointer to the message block from which data 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 VALUE

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

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

allocb(D3DK), **copymsg**(D3DK), **msgb**(D4DK)

EXAMPLE

This example illustrates how **copyb** can be used during message retransmission. If there are no messages to retransmit, we return (line 21). Otherwise, we lock the retransmission list (line 23). For each retransmission record in the list, we test to see if either the message has already been retransmitted, or if the downstream queue is full (by calling **canputnext**(D3DK) on line 26). If either is true, we skip the current retransmission record and continue searching the list. Otherwise, we use **copyb**(D3DK) to copy a header message block (line 30), and **dupmsg**(D3DK) to duplicate the data to be retransmitted (line 32).

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 37), link the message to be retransmitted to it (line 38), mark the retransmission record as having sent the message (line 39), unlock the retransmission list (line 40), and send the message downstream (line 41). Then we go back and lock the list again and start searching for more messages to retransmit.

This continues until we are either at the end of the retransmission list, or unable to send a message because of allocation failure. With the list still locked, we clear all the flags for sent messages (lines 44 and 45). Finally, we unlock the list lock and reschedule a **timeout** at the next valid interval (line 47) and return. Since we are using **itimeout**(D3DK), **retransmit** will run at the specified processor

NAME

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

SYNOPSIS

```
#include <sys/types.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*. 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.

RETURN VALUE

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

LEVEL

Base Only.

NOTES

May sleep.

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

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.

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.

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

SEE ALSO

bcopy(D3DK), **copyout**(D3DK), **uiomove**(D3DK), **ureadc**(D3DK), **uwritec**(D3DK)

EXAMPLE

A driver **ioctl**(D2DK) 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.

NAME

copymsg – copy a message

SYNOPSIS

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

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

ARGUMENTS

mp Pointer to the message to be copied.

DESCRIPTION

copymsg forms a new message by allocating new message blocks, copies the contents of the message referred to by *mp* (using the **copyb**(D3DK) function), and returns a pointer to the new message.

RETURN VALUE

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

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

allocb(D3DK), **copyb**(D3DK), **msgb**(D4DK)

EXAMPLE

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      }
```

NAME

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

SYNOPSIS

```
#include <sys/types.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.*

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.

RETURN VALUE

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

LEVEL

Base Only.

NOTES

May sleep.

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

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.

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.

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

SEE ALSO

bcopy(D3DK), **copyin**(D3DK), **uiomove**(D3DK), **ureadc**(D3DK), **uwritec**(D3DK)

EXAMPLE

A driver **ioctl**(D2DK) 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.

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. 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**.

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 VALUE

datamsg returns 1 if the message is a data message and 0 if the message is any other type.

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

allocb(D3DK), **datab**(D4DK), **msgb**(D4DK), **messages**(D5DK)

EXAMPLE

The **put**(D2DK) routine enqueues all data messages for handling by the **srv**(D2DK) (service) routine. All non-data messages are handled in the **put** routine.

```
1  xxxput(q, mp)
2      queue_t *q;
3      mblk_t *mp;
4  {
5      if (datamsg(mp->b_datap->db_type)) {
6              putq(q, mp);
7              return;
8      }
9      switch (mp->b_datap->db_type) {
10     case M_FLUSH:
       ...
11     }
12 }
```

NAME

dma_pageio – break up an I/O request into manageable units

SYNOPSIS

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

void dma_pageio(void (*strat)(), buf_t *bp);
```

ARGUMENTS

strat Address of the **strategy**(D2DK) routine to call to complete the I/O transfer.

bp Pointer to the buffer header structure.

DESCRIPTION

dma_pageio breaks up a data transfer request from **physiock**(D3DK) into units of contiguous memory. This function enhances the capabilities of the direct memory access controller (DMAC).

RETURN VALUE

None.

LEVEL

Base Only.

NOTES

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.

When the transfer completes, any allocated buffers are freed.

The interrupt priority level is not maintained across calls to **dma_pageio**.

SEE ALSO

read(D2DK), **strategy**(D2DK), **write**(D2DK), **physiock**(D3DK), **buf**(D4DK)

EXAMPLE

The following example shows how **dma_pageio** is used when reading or writing disk data. The driver's **read**(D2DK) and **write**(D2DK) entry points use **physiock** to check the validity of the I/O and perform the data transfer. The **strategy**(D2DK) routine passed to **physiock** just calls **dma_pageio** to perform the data transfer one page at a time.

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. Possible values 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**(D3DK) and **drv_usectohz**(D3DK) functions can be used, as necessary, to convert between clock ticks and microseconds (implementation independent units).

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**(D3D). 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**(D3DK). Since this value is associated with the current process, the caller must have process context (i.e. 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.

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.

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

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.

Several functions either take time values expressed in clock ticks as arguments [**itimeout**(D3DK), **delay**(D3DK)] or return time values expressed in clock ticks [**drv_getparm**(D3DK)]. 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**(D3DK) can be used, as necessary, to convert between clock ticks and microseconds.

RETURN VALUE

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 will be 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.

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.

SEE ALSO

delay(D3DK), **drv_getparm**(D3DK), **drv_usectohz**(D3DK), **dtimeout**(D3D), **itimeout**(D3DK)

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. Possible values are:

- **SYSCANC** Add *value* to the 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 the count of the number of modem interrupts received.
- **SYSOUTC** Add *value* to the count of the number of characters output to a terminal device.
- **SYSRAWC** Add *value* to the count of the number of characters received from a terminal device, before canonical processing has occurred.
- **SYSRINT** Add *value* to the count of the number of interrupts generated by data ready to be received from a terminal device.
- **SYSXINT** Add *value* to the count of the number of interrupts generated by data ready to be transmitted to a terminal device.

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*.

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

RETURN VALUE

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

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

drv_getparm(D3DK)

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. **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**(D3DK) 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.

RETURN VALUE

None.

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.

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.

SEE ALSO

delay(D3DK), **drv_hztousec**(D3DK), **drv_usectohz**(D3DK), **itimeout**(D3DK), **untimeout**(D3DK)

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.

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).

SEE ALSO

itimeout(D3DK), **LOCK_ALLOC**(D3DK), **untimeout**(D3DK)

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

SEE ALSO

copyb(D3DK), **dupmsg**(D3DK), **datab**(D4DK), **msgb**(D4DK)

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**(D3DK) function on *q*.

RETURN VALUE

None.

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.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

SEE ALSO

srv(D2DK), **noenable**(D3DK), **qenable**(D3DK), **queue**(D4DK)

EXAMPLE

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**(D3DK).

```
1  void
2  qrestart(q)
3      register queue_t *q;
4  {
5      enableok(q);
6      if (q->q_first)
7              qenable(q);
8  }
```

NAME

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

SYNOPSIS

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

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

ARGUMENTS

pri Priority of the **esballoc**(D3DK) allocation request (**BPRI_LO**, **BPRI_MED**, or **BPRI_HI**.)

func Function to be called when a buffer becomes available.

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

DESCRIPTION

esbbcall, like **bufcall**(D3DK), serves as a **timeout** call of indeterminate length. If **esballoc**(D3DK) is unable to allocate a message block header and a data block header to go with its externally supplied data buffer, **esbbcall** can be used to schedule the routine *func*, to be called with the argument *arg* when memory becomes available.

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

RETURN VALUE

If successful, **esbbcall** returns a non-zero value that identifies the scheduling request. This non-zero identifier may be passed to **unbufcall**(D3DK) to cancel the request. If any failure occurs, **esbbcall** returns 0.

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.

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**.

SEE ALSO

allocb(D3DK), **bufcall**(D3DK), **esballoc**(D3DK), **itimeout**(D3DK), **unbufcall**(D3DK)

NAME

flushband – flush messages in a specified priority band

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.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. Valid *flag* values are:

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

FLUSHALL Flush all messages.

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*.

RETURN VALUE

None.

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.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

SEE ALSO

put(D2DK), **flushq**(D3DK), **queue**(D4DK)

EXAMPLE

See **put**(D2DK) for an example of **flushband**.

NAME

freeb – free a message block

SYNOPSIS

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

void freeb(mblk_t *bp);
```

ARGUMENTS

bp Pointer to the message block to be deallocated.

DESCRIPTION

freeb deallocates a message block. If the reference count of the **db_ref** member of the **datab**(D4DK) 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**(D3DK), the driver is notified that the attached data buffer needs to be freed by calling the free-routine [see **free_rtn**(D4DK)] associated with the data buffer. Once this is accomplished, **freeb** releases the STREAMS resources associated with the buffer.

RETURN VALUE

None.

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

allocb(D3DK), **dupb**(D3DK), **esballoc**(D3DK), **datab**(D4DK), **free_rtn**(D4DK), **msgb**(D4DK)

EXAMPLE

See **copyb**(D3DK) for an example of **freeb**.

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**(D3DK). It may not be used on a buffer header obtained through any other interface. It is typically called from a driver's *iodone handler*, specified in the **b_iodone** field of the **buf**(D4DK) structure.

RETURN VALUE

None.

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

biodone(D3DK), **biowait**(D3DK), **getrbuf**(D3DK), **buf**(D4DK)

NAME

geteblk – get an empty buffer

SYNOPSIS

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

buf_t *geteblk();
```

DESCRIPTION

geteblk retrieves a buffer [see **buf**(D4DK)] from the buffer cache and returns a pointer to the buffer header. If a buffer is not available, **geteblk** sleeps until one is available.

When the driver **strategy**(D2DK) 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**(D2DK) 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**(D3DK). Or the driver can choose to use its own memory for the buffer and only allocate a buffer header with **getrbuf**(D3DK).

RETURN VALUE

A pointer to the buffer header structure is returned.

LEVEL

Base Only.

NOTES

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.

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*. External major numbers are visible to the user. Internal major numbers are only visible in the kernel. Since 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.

RETURN VALUE

The external major number.

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

etoimajor(D3DK), **geteminor**(D3DK), **getmajor**(D3DK), **getminor**(D3DK), **makedevice**(D3DK)

NAME

geterror – retrieve error number from a buffer header

SYNOPSIS

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

int geterror(struct buf *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**(D4DK) structure).

RETURN VALUE

An error number indicating the error condition of the I/O request is returned. If the I/O request completed successfully, 0 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

buf(D4DK), **errnos**(D5DK)

NAME

getminor – get internal minor device number

SYNOPSIS

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

minor_t getminor(dev_t dev);
```

ARGUMENTS

dev Device number.

DESCRIPTION

The **getminor** function extracts the internal minor number from a device number. See **getemajor**(D3DK) for an explanation of external and internal major numbers.

RETURN VALUE

The internal minor number.

LEVEL

Base or Interrupt.

NOTES

No validity checking is performed. If *dev* is invalid, an invalid number is returned.

Does not sleep.

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

SEE ALSO

etoimajor(D3DK), **getemajor**(D3DK), **geteminor**(D3DK), **getmajor**(D3DK), **makedevice**(D3DK)

NAME

getq – get the next message from a queue

SYNOPSIS

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

mblk_t *getq(queue_t *q);
```

ARGUMENTS

q Pointer to the queue from which the message is to be retrieved.

DESCRIPTION

getq is used by service [see **srv**(D2DK)] routines to retrieve queued messages. It gets the next available message from the top of the queue pointed to by *q*. **getq** handles flow control, restarting I/O that was blocked as needed.

RETURN VALUE

If there is a message to retrieve, **getq** returns a pointer to it. If no message is queued, **getq** returns a **NULL** pointer.

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.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

SEE ALSO

srv(D2DK), **bcanput**(D3DK), **canput**(D3DK), **putbq**(D3DK), **putq**(D3DK), **qenable**(D3DK), **rmvq**(D3DK)

EXAMPLE

See **srv**(D2DK) for an example of **getq**.

LEVEL

Base only if *flag* is set to **KM_SLEEP**. Base or interrupt if *flag* is set to **KM_NOSLEEP**.

NOTES

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*.

SEE ALSO

biodone(D3DK), **biowait**(D3DK), **freerbuf**(D3DK), **buf**(D4DK)

NAME

inl – read a 32 bit word from a 32 bit I/O port

SYNOPSIS

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

ulong_t inl(int port);
```

ARGUMENTS

port A valid 32 bit I/O port.

DESCRIPTION

This function provides a C language interface to the machine instruction that reads a 32 bit word from a 32 bit I/O port using the I/O address space, instead of the memory address space.

RETURN VALUE

Returns the value of the 32 bit word read from the I/O port.

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.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

SEE ALSO

Programmer's Reference Manual

Integrated Software Development Guide

inb(D3D), **inw**(D3D), **outb**(D3D), **outl**(D3D), **outw**(D3D), **repinsb**(D3D), **repinsd**(D3D), **repinsw**(D3D), **repoutsb**(D3D), **repoutsd**(D3D), **repoutsw**(D3D)

message (line 13), we free the entire message using **freemsg**(D3DK). Otherwise, for every **M_PROTO** message block in the message, we strip the **M_PROTO** block off using **unlinkb**(D3DK) (line 17) and free the message block using **freeb**(D3DK). When the header has been stripped, the data portion of the message is inserted back into the queue where it was originally found (line 21). Finally, when we are done searching the queue, we unfreeze the stream (line 26).

```
 1  void
 2  striproto(q)
 3     queue_t *q;
 4  {
 5     register mblk_t *emp, *nmp, *mp;
 6     pl_t pl;

 7     pl = freezestr(q);
 8     mp = q->q_first;
 9     while (mp) {
10            emp = mp->b_next;
11            if (mp->b_datap->db_type == M_PROTO) {
12                   rmvq(q, mp);
13                   if (msgdsize(mp) == 0) {
14                          freemsg(mp);
15                   } else {
16                          while (mp->b_datap->db_type == M_PROTO) {
17                                 nmp = unlinkb(mp);
18                                 freeb(mp);
19                                 mp = nmp;
20                          }
21                          insq(q, emp, mp);
22                   }
23            }
24            mp = emp;
25     }
26     unfreezestr(q, pl);
27  }
```

NAME

itimeout – execute a function after a specified length of time

SYNOPSIS

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

toid_t itimeout(void (*fn)(), void *arg, long ticks, pl_t pl);
```

ARGUMENTS

fn Function to execute when the time increment expires.

arg 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. *pl* must specify a priority level greater than or equal to *pltimeout*; thus, *plbase* cannot be used. See **LOCK_ALLOC**(D3DK) for a list of values for *pl*.

DESCRIPTION

itimeout causes the function specified by *fn* 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 *fn*. The **itimeout** call returns immediately without waiting for the specified function to execute.

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 *fn* must neither sleep nor reference process context.

RETURN VALUE

If the function specified by *fn* 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.

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.

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

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 **itimeout** 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.

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).

NAME

itoemajor – convert internal to external major device number

SYNOPSIS

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

major_t itoemajor(major_t imaj, major_t 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 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. 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**(D3DK) for an explanation of external and internal major numbers.

RETURN VALUE

External major number, or **NODEV**, if all have been searched.

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

etoimajor(D3DK), **getemajor**(D3DK), **geteminor**(D3DK), **getmajor**(D3DK), **getminor**(D3DK), **makedevice**(D3DK)

NAME

kmem_free – free previously allocated kernel memory

SYNOPSIS

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

void kmem_free(void *addr, size_t size);
```

ARGUMENTS

addr Address of the allocated memory to be returned. *addr* must specify the same address that was returned by the corresponding call to **kmem_alloc**(D3DK) or **kmem_zalloc**(D3DK) which allocated the memory.

size Number of bytes to free. The *size* parameter must specify the same number of bytes as was allocated by the corresponding call to **kmem_alloc** or **kmem_zalloc**.

DESCRIPTION

kmem_free returns *size* bytes of previously allocated kernel memory. 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).

RETURN VALUE

None.

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

kmem_alloc(D3DK), **kmem_zalloc**(D3DK)

NAME

kvtoppid – get physical page ID for kernel virtual address

SYNOPSIS

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

ppid_t kvtoppid(caddr_t addr);
```

ARGUMENTS

addr The kernel virtual address for which the physical page ID is to be returned.

DESCRIPTION

This routine can be used to obtain a physical page ID suitable to be used as the return value of the driver's **mmap**(D2DK) entry point. **kvtoppid** returns the physical page ID corresponding to the virtual address *addr*.

A physical page ID is a machine-specific token that uniquely identifies a page of physical memory in the system (either system memory or device memory.) No assumptions should be made about the format of a physical page ID.

RETURN VALUE

If *addr* is valid, the corresponding physical page ID is returned. Otherwise, **NOPAGE** 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

mmap(D2DK), **intro**(D3DK), **phystoppid**(D3D)

NAME

LOCK – acquire a basic lock

SYNOPSIS

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

pl_t 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. 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**(D3DK) for additional discussion and a list of the valid values for *pl*. Implementations which do not require that the interrupt priority level be raised during lock acquisition may choose to ignore this argument.

DESCRIPTION

LOCK sets the interrupt priority level in accordance with the value specified by *pl* (if required by the implementation) and acquires the lock specified by *lockp*. If the lock is not immediately available, the caller will wait until the lock is available. It is implementation defined whether the caller will block during the wait. Some implementations may cause the caller to spin for the duration of the wait, while on others the caller may block at some point.

RETURN VALUE

Upon acquiring the lock, **LOCK** returns the previous interrupt priority level (**plbase** - **plhi**).

LEVEL

Base or Interrupt.

NOTES

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 defined by the lock hierarchy [see **LOCK_ALLOC**(D3DK)] in order to avoid deadlock.

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 and recursion restrictions described above.

When called from interrupt level, the *pl* argument must not specify a priority level below the level at which the interrupt handler is running.

SEE ALSO

LOCK_ALLOC(D3DK), **LOCK_DEALLOC**(D3DK), **TRYLOCK**(D3DK), **UNLOCK**(D3DK)

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*.

lkinfop Pointer to a **lkinfo**(D4DK) structure. The **lk_name** member of the **lkinfo** structure points to a character string defining a name that will be associated with the lock for the purpose of statistics gathering. The name should begin with the driver prefix and should be unique to the lock or group of locks for which the driver wishes to collect a uniquely identifiable set of statistics (i.e. if a given name is shared by a group of locks, the statistics of individual locks within the group will not be uniquely identifiable). There are no flags defined within the **lk_flags** member of the **lkinfo** structure for use with **LOCK_ALLOC**.

The *lkinfop* pointer is recorded in a statistics buffer along with the lock statistics when the driver is compiled with the **DEBUG** and **_MPSTATS** compilation options defined. A given **lkinfo** structure may be shared among multiple basic locks and read/write locks but a **lkinfo** structure may not be shared between a basic lock and a sleep lock. The caller must ensure that the **lk_flags** and **lk_pad** members of the **lkinfo** structure are zeroed out before passing it to **LOCK_ALLOC**.

flag Specifies whether the caller is willing to sleep waiting for memory. 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. Under the **_MPSTATS** compilation option, if **KM_NOSLEEP** is specified and sufficient memory can be immediately allocated for the lock itself but not for an accompanying statistics buffer, **LOCK_ALLOC** will return a pointer to the allocated lock but individual statistics will not be collected for the lock.

NAME

LOCK_DEALLOC – deallocate an instance of a basic lock

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ksynch.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 VALUE

None.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

Attempting to deallocate a lock that is currently locked or is being waited for is an error and will result in undefined behavior.

Driver defined basic locks (other than the one being deallocated), read/write locks, and sleep locks may be held across calls to this function.

SEE ALSO

LOCK(D3DK), **LOCK_ALLOC**(D3DK), **TRYLOCK**(D3DK), **UNLOCK**(D3DK)

```
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      }
        ...
```

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. If the *int1* and *int2* arguments are not of the specified type the results are undefined.

Also, this interface may be implemented in a way that causes the arguments to be evaluated multiple times, so callers should beware of side effects.

RETURN VALUE

The lesser of the two integers.

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

max(D3DK)

NAME

msgpullup – concatenate bytes in a message

SYNOPSIS

```
#include <sys/stream.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. 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 VALUE

On success, a pointer to the new message is returned; on failure, **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

allocb(D3DK), **msgb**(D4DK)

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

Buffers allocated via **ngeteblk** must be freed using either **brelse**(D3DK) or **biodone**(D3DK).

SEE ALSO

biodone(D3DK), **biowait**(D3DK), **brelse**(D3DK), **geteblk**(D3DK), **buf**(D4DK)

NAME

OTHERQ – get pointer to queue's partner queue

SYNOPSIS

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

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

ARGUMENTS

q Pointer to the queue.

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. If *q* points to the read queue the write queue will be returned, and vice versa.

RETURN VALUE

OTHERQ returns a pointer to a queue's partner.

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

RD(D3DK), **WR**(D3DK)

EXAMPLE

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.

```
 1  void
 2  set_q_params(queue_t *q, long min, long max, ulong_t hi, ulong_t lo)
 3  {
 4      register pl_t pl;

 5      pl = freezestr(q);
 6      (void) strqset(q, QMINPSZ, 0, min);
 7      (void) strqset(q, QMAXPSZ, 0, max);
 8      (void) strqset(q, QHIWAT, 0, hi);
 9      (void) strqset(q, QLOWAT, 0, lo);
10      (void) strqset(OTHERQ(q), QMINPSZ, 0, min);
11      (void) strqset(OTHERQ(q), QMAXPSZ, 0, max);
12      (void) strqset(OTHERQ(q), QHIWAT, 0, hi);
13      (void) strqset(OTHERQ(q), QLOWAT, 0, lo);
14      unfreezestr(q, pl);
15  }
```

NAME

outl – write a 32 bit long word to a 32 bit I/O port

SYNOPSIS

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

void outl(int port, ulong_t data);
```

ARGUMENTS

port A valid 32 bit I/O port.

data The 32 bit value to be written to the port.

DESCRIPTION

This function provides a C language interface to the machine instruction that writes a 32 bit long word to a 32 bit I/O port using the I/O address space, instead of the memory address space.

RETURN VALUE

None.

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.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

SEE ALSO

Programmer's Reference Manual

Integrated Software Development Guide

inb(D3D), **inl**(D3D), **inw**(D3D), **outb**(D3D), **outw**(D3D), **repinsb**(D3D), **repinsd**(D3D), **repinsw**(D3D), **repoutsb**(D3D), **repoutsd**(D3D), **repoutsw**(D3D)

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.) The **db_type** field of the **datab**(D4DK) structure contains the message type. This field may be accessed through the message block using **mp->b_datap->db_type**.

RETURN VALUE

pcmsg returns 1 if the message is a priority control message and 0 if the message is any other type.

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

allocb(D3DK), **datab**(D4DK), **msgb**(D4DK), **messages**(D5DK)

EXAMPLE

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.

```
1     xxxsrv(q)
2           queue_t *q;
3     {
4           mblk_t *mp;

5           while ((mp = getq(q)) != NULL) {
6                 if (pcmsg(mp->b_datap->db_type) || canputnext(q)) {
7                       /* process message */
8                       putnext(q, mp);
9                 } else {
10                      putbq(q, mp);
11                      return;
12                }
13          }
14    }
```

NAME

phfree – free a pollhead structure

SYNOPSIS

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

void phfree(struct pollhead *php);
```

ARGUMENTS

php Pointer to the **pollhead** structure to be freed. The structure pointed to by *php* must have been previously allocated by a call to **phalloc**(D3DK).

DESCRIPTION

phfree frees the **pollhead** structure specified by *php*.

RETURN VALUE

None.

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.

DDI/DKI conforming drivers may 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.

SEE ALSO

chpoll(D2DK), **phalloc**(D3DK)

RETURN VALUE

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 **ENXIO** 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. **ENXIO** 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 pages for DMA.

LEVEL

Base Only.

NOTES

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.

Some device drivers need *nblocks* to be arbitrarily large (for example, for tapes whose sizes are unknown). In this case, *nblocks* should be no larger than (2^{22})-1.

SEE ALSO

ioctl(D2DK), **read**(D2DK), **strategy**(D2DK), **write**(D2DK), **dma_pageio**(D3D), **buf**(D4DK), **uio**(D4DK)

EXAMPLE

See **dma_pageio**(D3D) for an example of **physiock**.

NAME

physmap_free – free virtual address mapping for physical addresses

SYNOPSIS

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

void physmap_free(caddr_t vaddr, ulong_t nbytes, uint_t flags);
```

ARGUMENTS

vaddr Virtual address for which the mapping will be released.

nbytes Number of bytes in the mapping.

flags For future use (must be set to 0.)

DESCRIPTION

physmap_free releases a mapping allocated by a previous call to **physmap**. The *nbytes* argument must be identical to that given to **physmap**. Currently, no flags are supported and the *flags* argument must be set to zero. Generally, **physmap_free** will never be called, since drivers usually keep the mapping forever, but it is provided if a driver wants to dynamically allocate and free mappings.

RETURN VALUE

None.

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

physmap(D3D)

NAME

pollwakeup – inform polling processes that an event has occurred

SYNOPSIS

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

void pollwakeup(struct pollhead *php, short event);
```

ARGUMENTS

php Pointer to a **pollhead** structure.

event Event to notify the process about.

DESCRIPTION

The **pollwakeup** function provides non-STREAMS character drivers with a way to notify processes polling for the occurrence of an event. **pollwakeup** should be called from the driver for each occurrence of an event. Events are described in **chpoll**(D2DK).

The **pollhead** structure will usually be associated with the driver's private data structure for the particular minor device where the event has occurred.

RETURN

None.

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.

pollwakeup should only be called with one event at a time.

SEE ALSO

chpoll(D2DK)

poll(2) in the *Programmer's Reference Manual*

NAME

proc_ref – obtain a reference to a process for signaling

SYNOPSIS

```
void *proc_ref();
```

DESCRIPTION

A non-STREAMS character driver can call **proc_ref** to obtain a reference to the process in whose context it is running. The value returned can be used in subsequent calls to **proc_signal**(D3DK) to post a signal to the process. The return value should not be used in any other way (i.e. the driver should not attempt to interpret its meaning.)

RETURN VALUE

An identifier that can be used in calls to **proc_signal** and **proc_unref**(D3DK).

LEVEL

Base only.

NOTES

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**(D2DK) processing.

Requires user context.

Does not sleep.

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

SEE ALSO

proc_signal(D3DK), **proc_unref**(D3DK)

NAME

proc_unref – release a reference to a process

SYNOPSIS

```
void proc_unref(void *pref);
```

ARGUMENTS

pref Identifier obtained by a previous call to **proc_ref**(D3DK).

DESCRIPTION

The **proc_unref** function can be used to release a reference to a process identified by the parameter *pref*. There must be a matching call to **proc_unref** for every previous call to **proc_ref**(D3DK).

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

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.

Does not sleep.

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

SEE ALSO

proc_ref(D3DK), **proc_signal**(D3DK)

NAME

put – call a put procedure

SYNOPSIS

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

void put(queue_t *q, mblk_t *mp);
```

ARGUMENTS

q Pointer to a message queue.

mp Pointer to the message block being passed.

DESCRIPTION

put calls the put procedure (**put**(D2DK) entry point) for the queue specified by *q*, passing it the arguments *q* and *mp*. It is typically used by a driver or module to call its own put procedure so that the proper accounting is done in the stream.

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

DDI/DKI conforming drivers and modules are no longer permitted to call put procedures directly, but must call through the appropriate STREAMS utility function—for example, **put**(D3DK), **putnext**(D3DK), **putctl**(D3DK), **putnextctl**(D3DK), **qreply**(D3DK). **put(q, mp)** is provided as a DDI/DKI-conforming equivalent to a direct call to a **put** procedure, which is no longer allowed.

SEE ALSO

put(D2DK), **putctl**(D3DK), **putctl1**(D3DK), **putnext**(D3DK), **putnextctl**(D3DK), **putnextctl1**(D3DK), **qreply**(D3DK)

NAME

putctl – send a control message to a queue

SYNOPSIS

```
#include <sys/stream.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 control).

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**(D2DK) routine of the queue pointed to by *q*, passing it the allocated message.

RETURN VALUE

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.

NOTES

Does not sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The *q* argument to **putctl** and **putnextctl**(D3DK) may not reference **q_next** (e.g. an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI). **putnextctl(q, type)** is provided as a multiprocessor-safe equivalent to the common call **putctl(q->q_next, type)**, which is no longer allowed.

SEE ALSO

put(D2DK), **put**(D3DK), **putctl1**(D3DK), **putnextctl**(D3DK), **putnextctl1**(D3DK)

EXAMPLE

The **pass_ctl** routine is used to pass control messages to one's own queue. **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, **pass_ctl** returns 0, indicating failure (line 13).

```
1  int
2  pass_ctl(wrq, type, param)
3      queue_t *wrq;
4      uchar_t type;
5      uchar_t param;
```

NAME

putctl1 – send a control message with a one-byte parameter to a queue

SYNOPSIS

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

int putctl1(queue_t *q, int type, int param);
```

ARGUMENTS

q	Pointer to the queue to which the message is to be sent.
type	Message type (must be control).
param	One-byte parameter.

DESCRIPTION

putctl1, like **putctl**(D3DK), 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**(D2DK) routine of the queue pointed to by *q*, passing it the allocated message.

RETURN VALUE

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.

NOTES

Does not sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The *q* argument to **putctl1** and **putnextctl1**(D3DK) may not reference **q_next** (e.g. an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI). **putnextctl1(q, type, param)** is provided as a multiprocessor-safe equivalent to the common call **putctl1(q->q_next, type, param)**, which is no longer allowed.

SEE ALSO

put(D2DK), **put**(D3DK), **putctl**(D3DK), **putnextctl**(D3DK), **putnextctl1**(D3DK)

EXAMPLE

See **putctl**(D3DK) for an example of **putctl1**.

NAME

putnextctl – send a control message to a queue

SYNOPSIS

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

int putnextctl(queue_t *q, int type);
```

ARGUMENTS

q Pointer to the queue from which the message is to be sent.

type Message type (must be control type).

DESCRIPTION

putnextctl tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. **putnextctl** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putnextctl** calls the **put**(D2DK) procedure of the queue pointed to by *q->q_next*, passing it the allocated message.

RETURN VALUE

Upon successful completion, **putnextctl** returns 1. If *type* is a data type, or if a message block cannot be allocated, 0 is returned.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The *q* argument to **putctl**(D3DK) and **putnextctl** may not reference **q_next** (for example, an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI). **putnextctl(q, type)** is provided as a multiprocessor-safe equivalent to the common call **putctl(q->q_next, type)**, which is no longer allowed.

SEE ALSO

put(D2DK), **put**(D3DK), **putctl**(D3DK), **putctl1**(D3DK), **putnextctl1**(D3DK)

EXAMPLE

The **send_ctl** routine is used to pass control messages downstream. **M_BREAK** messages are handled with **putnextctl** (line 9). **putnextctl1** (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  {
```

NAME

putnextctl1 – send a control message with a one byte parameter to a queue

SYNOPSIS

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

int putnextctl1(queue_t *q, int type, int param);
```

ARGUMENTS

q	Pointer to the queue from which the message is to be sent.
type	Message type (must be control type).
param	One byte parameter.

DESCRIPTION

putnextctl1 tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. **putnextctl1** fails if *type* is **M_DATA**, **M_PROTO**, or **M_PCPROTO**, or if a message block cannot be allocated. If successful, **putnextctl** calls the **put**(D2DK) procedure of the queue pointed to by *q->q_next*, passing it the allocated message with the one byte parameter specified by *param*.

RETURN VALUE

Upon successful completion, **putnextctl1** returns 1. If *type* is a data type, or if a message block cannot be allocated, 0 is returned.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The *q* argument to **putctl1**(D3DK) and **putnextctl1** may not reference **q_next** (for example, an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI). **putnextctl1(q, type, param)** is provided as a multiprocessor-safe equivalent to the common call **putctl1(q->q_next, type, param)**, which is no longer allowed.

SEE ALSO

put(D2DK), **put**(D3DK), **putctl**(D3DK), **putctl1**(D3DK), **putnextctl**(D3DK)

EXAMPLE

See **putnextctl**(D3DK) for an example of **putnextctl1**.

NAME

qenable – schedule a queue's service routine to be run

SYNOPSIS

```
#include <sys/stream.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. **qenable** works regardless of whether the service routine has been disabled by a previous call to **noenable**(D3DK).

RETURN VALUE

None.

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.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

SEE ALSO

srv(D2DK), **enableok**(D3DK), **noenable**(D3DK), **queue**(D4DK)

EXAMPLE

See **enableok**(D3DK) for an example of the **qenable**.

NAME

qprocson – enable put and service routines

SYNOPSIS

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

void qprocson(queue_t *rq);
```

ARGUMENTS

rq Pointer to a read queue.

DESCRIPTION

qprocson enables the put and service routines of the driver or module whose read queue is pointed to by *rq*. Prior to the call to **qprocson**, the put and service routines of a newly pushed module or newly opened driver are disabled. For the module, messages flow around it as if it were not present in the stream.

qprocson must be called by the first open of a module or driver after allocation and initialization of any resources on which the put and service routines depend.

RETURN VALUE

None.

LEVEL

Base Level Only.

NOTES

May sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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.

SEE ALSO

open(D2DK), **put**(D2DK), **srv**(D2DK), **qprocsoff**(D3DK)

NAME

qsize – find the number of messages on a queue

SYNOPSIS

```
#include <sys/stream.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 VALUE

If there are no message on the queue, **qsize** returns 0. Otherwise, it returns the number of messages on the queue.

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.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

SEE ALSO

msgb(D4DK), **queue**(D4DK)

NAME

repinsb – read bytes from I/O port to buffer

SYNOPSIS

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

void repinsb(int port, uchar_t *addr, int cnt);
```

ARGUMENTS

port A valid 8 bit I/O port.

addr The address of the buffer where data is stored after *cnt* reads of the I/O port.

cnt The number of bytes to be read from the I/O port.

DESCRIPTION

This function provides a C language interface to the machine instructions that read a string of bytes from an 8 bit I/O port using the I/O address space, instead of the memory address space. The data from *cnt* reads of the I/O port is stored in the data buffer pointed to by *addr*. The data buffer should be at least *cnt* bytes in length.

RETURN VALUE

None.

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.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

SEE ALSO

Programmer's Reference Manual

Integrated Software Development Guide

inb(D3D), **inl**(D3D), **inw**(D3D), **outb**(D3D), **outl**(D3D), **outw**(D3D), **repinsd**(D3D), **repinsw**(D3D), **repoutsb**(D3D), **repoutsd**(D3D), **repoutsw**(D3D)

NAME

repinsw – read 16 bit words from I/O port to buffer

SYNOPSIS

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

void repinsw(int port, ushort_t *addr, int cnt);
```

ARGUMENTS

port A valid 16 bit I/O port.

addr The address of the buffer where data is stored after *cnt* reads of the I/O port.

cnt The number of 16 bit words to be read from the I/O port.

DESCRIPTION

This function provides a C language interface to the machine instructions that read a string of 16 bit short words from a 16 bit I/O port using the I/O address space, instead of the memory address space. The data from *cnt* reads of the I/O port is stored in the data buffer pointed to by *addr*. The data buffer should be at least *cnt* 16 bit words in length.

RETURN VALUE

None.

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.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

SEE ALSO

Programmer's Reference Manual

Integrated Software Development Guide

inb(D3D), **inl**(D3D), **inw**(D3D), **outb**(D3D), **outl**(D3D), **outw**(D3D), **repinsb**(D3D), **repinsd**(D3D), **repoutsb**(D3D), **repoutsd**(D3D), **repoutsw**(D3D)

NAME

repoutsd – write 32 bit words from buffer to an I/O port

SYNOPSIS

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

void repoutsd(int port, ulong_t *addr, int cnt);
```

ARGUMENTS

port A valid 32 bit I/O port.

addr The address of the buffer from which *cnt* 32 bit words are written to the I/O port.

cnt The number of 32 bit words to be written to the I/O port.

DESCRIPTION

This function provides a C language interface to the machine instructions that write a string of 32 bit long words to a 32 bit I/O port using the I/O address space, instead of the memory address space. *cnt* 32 bit words starting at the address pointed to by *addr* are written to the I/O port in *cnt* write operations. The buffer should be at least *cnt* 32 bit words in length.

RETURN VALUE

None.

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.

This function may not be meaningful on all implementations because some implementations may not support I/O-mapped I/O.

SEE ALSO

Programmer's Reference Manual

Integrated Software Development Guide

inb(D3D), **inl**(D3D), **inw**(D3D), **outb**(D3D), **outl**(D3D), **outw**(D3D), **repinsb**(D3D), **repinsd**(D3D), **repinsw**(D3D), **repoutsb**(D3D), **repoutsw**(D3D)

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*. The map must have been allocated by a call to **rmallocmap**(D3DK) and the space managed by the map must have been added using **rmfree**(D3DK) 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**.

RETURN VALUE

Upon successful completion, **rmalloc** returns the base of the allocated space. If *size* units cannot be allocated, 0 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

rmalloc_wait(D3DK), **rmallocmap**(D3DK), **rmfree**(D3DK), **rmfreemap**(D3DK)

NAME

rmalloc_wait – allocate space from a private space management map

SYNOPSIS

```
#include <sys/types.h>
#include <sys/map.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**(D3DK). **rmalloc_wait** is identical to **rmalloc**(D3DK), except that a caller 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**(D3DK).

RETURN VALUE

rmalloc_wait returns the base of the allocated space.

LEVEL

Base Level Only.

NOTES

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.

SEE ALSO

rmalloc(D3DK), **rmallocmap**(D3DK), **rmfree**(D3DK), **rmfreemap**(D3DK)

NAME

rmfreemap – free a private space management map

SYNOPSIS

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

void rmfreemap(struct map *mp);
```

ARGUMENTS

mp Pointer to the map to be freed. The **map** structure array pointed to by *mp* must have been previously allocated by a call to **rmallocmap**(D3DK).

DESCRIPTION

rmfreemap frees the map pointed to by *mp*.

RETURN VALUE

None.

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.

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.

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.

SEE ALSO

rmalloc(D3DK), **rmalloc_wait**(D3DK), **rmallocmap**(D3DK), **rmfree**(D3DK)

NAME

rmvq – remove a message from a queue

SYNOPSIS

```
#include <sys/stream.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 a message from a queue. 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**(D3DK) should be used to remove a message from a queue.

RETURN VALUE

None.

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.

The caller must have the stream frozen [see **freezestr**(D3DK)] when calling this function.

mp must point to an existing message in the queue pointed to by *q*, or a system panic will occur.

SEE ALSO

freezestr(D3DK), **getq**(D3DK), **insq**(D3DK), **unfreezestr**(D3DK)

EXAMPLE

See **insq**(D3DK) for an example of **rmvq**.

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 **RW_ALLOC** call does not actually cause any interrupts to be blocked upon lock acquisition, it simply asserts that subsequent **RW_RDLOCK/RW_WRLOCK** calls to acquire this lock will pass in a priority level at least as great as *min_pl*.

lkinfop Pointer to a **lkinfo**(D4DK) structure. The **lk_name** member of the **lkinfo** structure points to a character string defining a name that will be associated with the lock for the purpose of statistics gathering. The name should begin with the driver prefix and should be unique to the lock or group of locks for which the driver wishes to collect a uniquely identifiable set of statistics (i.e. if a given name is shared by a group of locks, the statistics of individual locks within the group will not be uniquely identifiable). There are no flags defined within the **lk_flags** member of the **lkinfo** structure for use with **RW_ALLOC**.

The *lkinfop* pointer is recorded in a statistics buffer along with the lock statistics when the driver is compiled with the **DEBUG** and **_MPSTATS** compilation options defined. A given **lkinfo** structure may be shared among multiple read/write locks and basic locks but a **lkinfo** structure may not be shared between a read/write lock and a sleep lock. The caller must ensure that the **lk_flags** and **lk_pad** members of the **lkinfo** structure are zeroed out before passing it to **RW_ALLOC**.

flag Specifies whether the caller is willing to sleep waiting for memory. 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 **RW_ALLOC** will return **NULL** if sufficient memory is not immediately available. Under the **_MPSTATS** compilation option, if **KM_NOSLEEP** is specified and sufficient memory can be immediately allocated for the lock itself but not for an accompanying statistics buffer, **RW_ALLOC** will return a pointer to the allocated lock but individual statistics will not be collected for the lock.

NAME

RW_DEALLOC – deallocate an instance of a read/write lock

SYNOPSIS

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

void RW_DEALLOC(rwlock_t *lockp);
```

ARGUMENTS

lockp Pointer to the read/write lock to be deallocated.

DESCRIPTION

RW_DEALLOC deallocates the read/write lock specified by *lockp*.

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

Attempting to deallocate a lock that is currently locked or is being waited for is an error and will result in undefined behavior.

Driver defined locks, read/write locks (other than the one being deallocated), and sleep locks may be held across calls to this function.

SEE ALSO

RW_ALLOC(D3DK), **RW_RDLOCK**(D3DK), **RW_TRYRDLOCK**(D3DK), **RW_TRYWRLOCK**(D3DK), **RW_UNLOCK**(D3DK), **RW_WRLOCK**(D3DK)

SEE ALSO

RW_ALLOC(D3DK), RW_DEALLOC(D3DK), RW_TRYRDLOCK(D3DK), RW_TRYWRLOCK(D3DK), RW_UNLOCK(D3DK), RW_WRLOCK(D3DK)

NAME

RW_TRYWRLOCK – try to acquire a read/write lock in write mode

SYNOPSIS

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

pl_t RW_TRYWRLOCK(rwlock_t *lockp, pl_t pl);
```

ARGUMENTS

lockp Pointer to the read/write lock to be acquired.

pl The interrupt priority level to be set while the lock is held by the caller. 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 **RW_ALLOC**(D3DK) for additional discussion and a list of the valid values for *pl*. Implementations which do not require that the interrupt priority level be raised during lock acquisition may choose to ignore this argument.

DESCRIPTION

If the lock specified by *lockp* is immediately available in write mode (no context is holding the lock in read mode or write mode), **RW_TRYWRLOCK** sets the interrupt priority level in accordance with the value specified by *pl* (if required by the implementation) and acquires the lock in write mode. If the lock is not immediately available in write mode, the function returns without acquiring the lock.

RETURN VALUE

If the lock is acquired, **RW_TRYWRLOCK** returns the previous interrupt priority level (**plbase** - **plhi**). If the lock is not acquired the value **invpl** is returned.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

RW_TRYWRLOCK may be used to acquire a lock in a different order from the order defined by the lock hierarchy.

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

When called from interrupt level, the *pl* argument must not specify a priority level below the level at which the interrupt handler is running.

SEE ALSO

RW_ALLOC(D3DK), **RW_DEALLOC**(D3DK), **RW_RDLOCK**(D3DK), **RW_TRYRDLOCK**(D3DK), **RW_UNLOCK**(D3DK), **RW_WRLOCK**(D3DK)

NAME

RW_WRLOCK – acquire a read/write lock in write mode

SYNOPSIS

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

pl_t RW_WRLOCK(rwlock_t *lockp, pl_t pl);
```

ARGUMENTS

lockp Pointer to the read/write lock to be acquired.

pl The interrupt priority level to be set while the lock is held by the caller. 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 **RW_ALLOC**(D3DK) for additional discussion and a list of the valid values for *pl*. Implementations which do not require that the interrupt priority level be raised during lock acquisition may choose to ignore this argument.

DESCRIPTION

RW_WRLOCK sets the interrupt priority level in accordance with the value specified by *pl* (if required by the implementation) and acquires the lock specified by *lockp* in write mode. If the lock cannot be acquired immediately in write mode, the caller will wait until the lock is available in write mode. (A read/write lock is available in write mode when the lock is not held by any context). It is implementation defined whether the caller will block during the wait. Some implementations may cause the caller to spin for the duration of the wait, while on others the caller may block at some point.

RETURN VALUE

Upon acquiring the lock, **RW_WRLOCK** returns the previous interrupt priority level (**plbase** - **plhi**).

LEVEL

Base or Interrupt.

NOTES

Read/write locks are not recursive. A call to **LOCK** attempting to acquire a lock that is currently held by the calling context may result in deadlock.

Calls to **RW_WRLOCK** should honor the ordering defined by the lock hierarchy [see **RW_ALLOC**(D3DK)] in order to avoid deadlock.

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 and recursion restrictions described above.

When called from interrupt level, the *pl* argument must not specify a priority level below the level at which the interrupt handler is running.

NAME

SAMESTR – test if next queue is same type

SYNOPSIS

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

int SAMESTR(queue_t *q);
```

ARGUMENTS

q Pointer to the queue.

DESCRIPTION

The **SAMESTR** function is used to see if the next queue in a stream (if it exists) is the same type as the current queue (that is, both are read queues or both are write queues). This can be used to determine the point in a STREAMS-based pipe where a read queue is linked to a write queue.

RETURN VALUE

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

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

The caller cannot have the stream frozen [see **freezestr**(D3DK)] when calling this function.

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

The argument *q* may not reference **q_next** (for example, an argument of **q->q_next** is erroneous on a multiprocessor and is disallowed by the DDI/DKI).

SEE ALSO

OTHERQ(D3DK)

EXAMPLE

See the **put**(D2DK) manual page for an example of **SAMESTR**.

LEVEL

Base only if *flag* is set to **KM_SLEEP**. Base or interrupt if *flag* is set to **KM_NOSLEEP**.

NOTES

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*.

SEE ALSO

SLEEP_DEALLOC(D3DK), **SLEEP_LOCK**(D3DK), **SLEEP_LOCK_SIG**(D3DK), **SLEEP_LOCKAVAIL**(D3DK), **SLEEP_LOCKOWNED**(D3DK), **SLEEP_TRYLOCK**(D3DK), **SLEEP_UNLOCK**(D3DK), **lkinfo**(D4DK)

NAME

SLEEP_LOCK – acquire a sleep lock

SYNOPSIS

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

void SLEEP_LOCK(sleep_t *lockp, int priority);
```

ARGUMENTS

lockp Pointer to the sleep lock to be acquired.

priority A hint to the the scheduling policy as to the relative priority the caller wishes to be assigned while running in the kernel after waking up. The valid values for this argument are as follows:

pridisk	Priority appropriate for disk driver.
prinet	Priority appropriate for network driver.
pritty	Priority appropriate for terminal driver.
pritape	Priority appropriate for tape driver.
prihi	High priority.
primed	Medium priority.
prilo	Low priority.

Drivers may use these values to request a priority appropriate to a given type of device or to request a priority that is high, medium or low relative to other activities within the kernel.

It is also permissible to specify positive or negative offsets from the values defined above. Positive offsets result in more favorable priority. The maximum allowable offset in all cases is 3 (e.g. **pridisk+3** and **pridisk-3** are valid values but **pridisk+4** and **pridisk-4** are not valid). Offsets can be useful in defining the relative importance of different locks or resources that may be held by a given driver. 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**.

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.

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 VALUE

None.

NAME

SLEEP_LOCKAVAIL – query whether a sleep lock is available

SYNOPSIS

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

bool_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.

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.

RETURN VALUE

SLEEP_LOCKAVAIL returns **TRUE** (a non-zero value) if the lock was available or **FALSE** (zero) if the lock was not available.

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

SLEEP_ALLOC(D3DK), **SLEEP_DEALLOC**(D3DK), **SLEEP_LOCK**(D3DK), **SLEEP_LOCK_SIG**(D3DK), **SLEEP_LOCKOWNED**(D3DK), **SLEEP_TRYLOCK**(D3DK), **SLEEP_UNLOCK**(D3DK)

NAME

SLEEP_LOCK_SIG – acquire a sleep lock

SYNOPSIS

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

bool_t SLEEP_LOCK_SIG(sleep_t *lockp, int priority);
```

ARGUMENTS

lockp Pointer to the sleep lock to be acquired.

priority A hint to the the scheduling policy as to the relative priority the caller wishes to be assigned while running in the kernel after waking up. The valid values for this argument are as follows:

pridisk	Priority appropriate for disk driver.
prinet	Priority appropriate for network driver.
pritty	Priority appropriate for terminal driver.
pritape	Priority appropriate for tape driver.
prihi	High priority.
primed	Medium priority.
prilo	Low priority.

Drivers may use these values to request a priority appropriate to a given type of device or to request a priority that is high, medium or low relative to other activities within the kernel.

It is also permissible to specify positive or negative offsets from the values defined above. Positive offsets result in more favorable priority. The maximum allowable offset in all cases is 3 (e.g. **pridisk+3** and **pridisk-3** are valid values but **pridisk+4** and **pridisk-4** are not valid). Offsets can be useful in defining the relative importance of different locks or resources that may be held by a given driver. 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**.

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.

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.

NAME

SLEEP_TRYLOCK – try to acquire a sleep lock

SYNOPSIS

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

bool_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) **SLEEP_TRYLOCK** acquires the lock. If the lock is not immediately available, the function returns without acquiring the lock.

RETURN VALUE

SLEEP_TRYLOCK returns **TRUE** (a non-zero value) if the lock is successfully acquired or **FALSE** (zero) if the lock is not acquired.

LEVEL

Base Level Only.

NOTES

Does not sleep.

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

SEE ALSO

SLEEP_ALLOC(D3DK), **SLEEP_DEALLOC**(D3DK), **SLEEP_LOCK**(D3DK), **SLEEP_LOCK_SIG**(D3DK), **SLEEP_LOCKAVAIL**(D3DK), **SLEEP_LOCKOWNED**(D3DK), **SLEEP_UNLOCK**(D3DK)

NAME

spl – block/allow interrupts on a processor

SYNOPSIS

```
pl_t splbase();
pl_t spltimeout();
pl_t spldisk();
pl_t splstr();
pl_t splhi();
pl_t splx(pl_t 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. 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 or read/write locks for this purpose [see **LOCK_ALLOC**(D3DK) or **RW_ALLOC**(D3DK)].

The **spl** functions include the following:

splbase	Block no interrupts.
spltimeout	Block functions scheduled by **itimeout** and **dtimeout**.
spldisk	Block disk device interrupts.
splstr	Block STREAMS interrupts.
splhi	Block all interrupts.

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.

RETURN VALUE

All **spl** functions return the previous priority level.

NOTES

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.

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>

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. Valid 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 **printf**(3S) style format string. **%s**, **%e**, **%g**, and **%G** formats are not allowed.

args Zero or more arguments to **printf** (maximum of **NLOGARGS**, currently three).

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 VALUE

strlog returns 0 if the message is not seen by all the readers, **1** otherwise.

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.

NAME

strqget – get information about a queue or band of the queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.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. Valid 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**(D4DK)].

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 VALUE

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*.

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.

The caller must have the stream frozen [see **freezestr**(D3DK)] when calling this function.

SEE ALSO

freezestr(D3DK), **strqset**(D3DK), **unfreezestr**(D3DK), **queue**(D4DK)

NAME

SV_ALLOC – allocate and initialize a synchronization variable

SYNOPSIS

```
#include <sys/kmem.h>
#include <sys/ksynch.h>

sv_t *SV_ALLOC(int flag);
```

ARGUMENTS

flag Specifies whether the caller is willing to sleep waiting for memory. 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 **SV_ALLOC** will return **NULL** if sufficient memory is not immediately available.

DESCRIPTION

SV_ALLOC dynamically allocates and initializes an instance of a synchronization variable.

RETURN VALUE

Upon successful completion, **SV_ALLOC** returns a pointer to the newly allocated synchronization variable. If **KM_NOSLEEP** is specified and sufficient memory is not immediately available, **SV_ALLOC** returns a **NULL** pointer.

LEVEL

Base only if *flag* is set to **KM_SLEEP**. Base or interrupt if *flag* is set to **KM_NOSLEEP**.

NOTES

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*.

SEE ALSO

SV_BROADCAST(D3DK), **SV_DEALLOC**(D3DK), **SV_SIGNAL**(D3DK), **SV_WAIT**(D3DK), **SV_WAIT_SIG**(D3DK)

NAME

SV_DEALLOC – deallocate an instance of a synchronization variable

SYNOPSIS

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

void SV_DEALLOC(sv_t *svp);
```

ARGUMENTS

lockp Pointer to the synchronization variable to be deallocated.

DESCRIPTION

SV_DEALLOC deallocates the synchronization variable specified by *svp*.

RETURN VALUE

None.

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

SV_ALLOC(D3DK), **SV_BROADCAST**(D3DK), **SV_SIGNAL**(D3DK), **SV_WAIT**(D3DK), **SV_WAIT_SIG**(D3DK)

NAME

SV_WAIT – sleep on a synchronization variable

SYNOPSIS

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

void SV_WAIT(sv_t *svp, int priority, lock_t *lkp);
```

ARGUMENTS

svp Pointer to the synchronization variable on which to sleep.

priority A hint to the the scheduling policy as to the relative priority the caller wishes to be assigned while running in the kernel after waking up. The valid values for this argument are as follows:

pridisk	Priority appropriate for disk driver.
prinet	Priority appropriate for network driver.
pritty	Priority appropriate for terminal driver.
pritape	Priority appropriate for tape driver.
prihi	High priority.
primed	Medium priority.
prilo	Low priority.

Drivers may use these values to request a priority appropriate to a given type of device or to request a priority that is high, medium or low relative to other activities within the kernel.

It is also permissible to specify positive or negative offsets from the values defined above. Positive offsets result in more favorable priority. The maximum allowable offset in all cases is 3 (e.g. **pridisk+3** and **pridisk-3** are valid values but **pridisk+4** and **pridisk-4** are not valid). Offsets can be useful in defining the relative importance of different locks or resources that may be held by a given driver. In general, a higher relative priority should be used when the caller is sleeping waiting for a highly contended kernel resource, or when the caller is already holding one or more locks or kernel resources upon entry to **SV_WAIT**.

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.

lkp Pointer to a basic lock which must be locked when **SV_WAIT** is called. The basic lock is released when the calling process goes to sleep, as described below.

DESCRIPTION

SV_WAIT causes the calling process to go to sleep (the caller's execution is suspended and other processes may be scheduled) waiting for a call to **SV_SIGNAL**(D3DK) or **SV_BROADCAST**(D3DK) for the synchronization variable specified by *svp*.

NAME

SV_WAIT_SIG – sleep on a synchronization variable

SYNOPSIS

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

bool_t SV_WAIT_SIG(sv_t *svp, int priority, lock_t *lkp);
```

ARGUMENTS

svp Pointer to the synchronization variable on which to sleep.

priority A hint to the the scheduling policy as to the relative priority the caller wishes to be assigned while running in the kernel after waking up. The valid values for this argument are as follows:

pridisk	Priority appropriate for disk driver.
prinet	Priority appropriate for network driver.
pritty	Priority appropriate for terminal driver.
pritape	Priority appropriate for tape driver.
prihi	High priority.
primed	Medium priority.
prilo	Low priority.

Drivers may use these values to request a priority appropriate to a given type of device or to request a priority that is high, medium or low relative to other activities within the kernel.

It is also permissible to specify positive or negative offsets from the values defined above. Positive offsets result in more favorable priority. The maximum allowable offset in all cases is 3 (e.g. **pridisk+3** and **pridisk-3** are valid values but **pridisk+4** and **pridisk-4** are not valid). Offsets can be useful in defining the relative importance of different locks or resources that may be held by a given driver. In general, a higher relative priority should be used when the caller is sleeping waiting for a highly contended kernel resource, or when the caller is already holding one or more locks or kernel resources upon entry to **SV_WAIT_SIG**.

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.

lkp Pointer to a basic lock which must be locked when **SV_WAIT_SIG** is called. The basic lock is released when the calling process goes to sleep, as described below.

DESCRIPTION

SV_WAIT_SIG causes the calling process to go to sleep (the caller's execution is suspended and other processes may be scheduled) waiting for a call to **SV_SIGNAL**(D3DK) or **SV_BROADCAST**(D3DK) for the synchronization variable specified by *svp*.

NAME

TRYLOCK – try to acquire a basic lock

SYNOPSIS

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

pl_t 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. 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**(D3DK) for additional discussion and a list of the valid values for *pl*. Implementations which do not require that the interrupt priority level be raised during lock acquisition may choose to ignore this argument.

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* (if required by the implementation) and acquires the lock. If the lock is not immediately available, the function returns without acquiring the lock.

RETURN VALUE

If the lock is acquired, **TRYLOCK** returns the previous interrupt priority level (**plbase** - **plhi**). If the lock is not acquired the value **invpl** is returned.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

TRYLOCK may be used to acquire a lock in a different order from the order defined by the lock hierarchy.

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

When called from interrupt level, the *pl* argument must not specify a priority level below the level at which the interrupt handler is running.

SEE ALSO

LOCK(D3DK), **LOCK_ALLOC**(D3DK), **LOCK_DEALLOC**(D3DK), **UNLOCK**(D3DK)

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.

SEE ALSO

bcopy(D3DK), **copyin**(D3DK), **copyout**(D3DK), **ureadc**(D3DK), **uwritec**(D3DK), **iovec**(D4DK), **uio**(D4DK)

```
15      qprocsoff(q);
16      if (modp->m_type == BUFCALL)
17              unbufcall(modp->m_id);
18      else if (modp->m_type == TIMEOUT)
19              untimeout(modp->m_id);
20      modp->m_type = 0;
        . . .
```

NAME

unlinkb – remove a message block from the head of a message

SYNOPSIS

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

mblk_t *unlinkb(mblk_t *mp);
```

ARGUMENTS

mp Pointer to the message.

DESCRIPTION

unlinkb removes the first message block from the message pointed to by *mp*. The removed message block is not freed. It is the caller's responsibility to free it.

RETURN VALUE

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.

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

linkb(D3DK)

EXAMPLE

The routine expects to get passed an **M_PROTO T_DATA_IND** message. It will remove and free the **M_PROTO** header and return the remaining **M_DATA** portion of the message.

```
1  mblk_t *
2  makedata(mp)
3       mblk_t *mp;
4  {
5       mblk_t *nmp;

6       nmp = unlinkb(mp);
7       freeb(mp);
8       return(nmp);
9  }
```

NAME

untimeout – cancel previous **timeout** request

SYNOPSIS

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

void untimeout(toid_t id);
```

ARGUMENTS

id Identifier returned from a previous call to **dtimeout**(D3D) or **itimeout**(D3DK).

DESCRIPTION

untimeout cancels a pending **timeout** request. If the **untimeout** is called while the function is running, then **untimeout** will not return until the function has completed. The function that runs as a result of a call to **dtimeout** or **itimeout** cannot use **untimeout** to cancel itself.

RETURN VALUE

None.

LEVEL

Base or Interrupt, with the following exception: The **untimeout** can only be performed from interrupt levels less than, or equal to, the level specified when the function was scheduled.

NOTES

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.

SEE ALSO

delay(D3DK), **dtimeout**(D3D), **itimeout**(D3DK), **unbufcall**(D3DK)

EXAMPLE

See **unbufcall**(D3DK) for an example of **untimeout**.

NAME

uwritec – return a character from space described by **uio**(D4DK) structure

SYNOPSIS

```
#include <sys/uio.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**(D4DK) 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 VALUE

If successful, **uwritec** returns the character. **-1** is returned if the space described by the **uio** structure is empty or there is an error.

LEVEL

Base only if **uio_segflg** is set to **UIO_USERSPACE**. Base or interrupt if **uio_segflg** is set to **UIO_SYSSPACE**.

NOTES

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**.

When holding 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.

SEE ALSO

uiomove(D3DK), **ureadc**(D3DK), **iovec**(D4DK), **uio**(D4DK)

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 VALUE

The pointer to the write queue.

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

OTHERQ(D3DK), RD(D3DK)

EXAMPLE

In a STREAMS **open**(D2DK) 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**(D2DK) or **srv**(D2DK) routines.

```
1  extern struct xxx_dev[];
   ...
2  xxxopen(queue_t *q, dev_t *devp, int flag, int sflag, cred_t *crp)
3  {
     ...
3    q->q_ptr = (caddr_t)&xxx_dev[getminor(*devp)];
4    WR(q)->q_ptr = (caddr_t)&xxx_dev[getminor(*devp)];
     ...
5  }
```

NAME

dma_disable – disable recognition of hardware requests on a DMA channel

SYNOPSIS

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

void dma_disable(int chan);
```

ARGUMENTS

chan Channel to be disabled.

DESCRIPTION

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

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

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.

SEE ALSO

dma_enable(D3X), **dma_prog**(D3X), **dma_cb**(D4X)

NAME

dma_free_buf – free a previously allocated DMA buffer descriptor

SYNOPSIS

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

void dma_free_buf(struct dma_buf *dmabufptr);
```

ARGUMENTS

dmabufptr Address of the allocated DMA buffer descriptor to be returned.

DESCRIPTION

dma_free_buf frees a DMA buffer descriptor. The *dmabufptr* argument must specify the address of a DMA buffer descriptor previously allocated by **dma_get_buf**(D3X).

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

SEE ALSO

dma_get_buf(D3X), **dma_buf**(D4X)

NAME

dma_get_buf – allocate a DMA buffer descriptor

SYNOPSIS

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

struct dma_buf *dma_get_buf(uchar_t mode);
```

ARGUMENTS

mode Specifies whether the caller is willing to sleep waiting for memory. If *mode* is set to **DMA_SLEEP**, the caller will sleep if necessary until the memory for a **dma_buf** is available. If *mode* is set to **DMA_NOSLEEP**, the caller will not sleep, but **dma_get_buf** will return **NULL** if memory for a **dma_buf** is not immediately available.

DESCRIPTION

dma_get_buf allocates memory for a DMA command block structure [see **dma_buf**(D4X)], zeroes it out, and returns a pointer to the structure.

RETURN VALUE

dma_get_buf returns a pointer to the allocated DMA control block. If **DMA_NOSLEEP** is specified and memory for a **dma_buf** is not immediately available, **dma_get_buf** returns a **NULL** pointer.

LEVEL

Base only if *mode* is set to **DMA_SLEEP**. Base or Interrupt if *mode* is set to **DMA_NOSLEEP**.

NOTES

Can sleep if *mode* is set to **DMA_SLEEP**.

SEE ALSO

dma_free_buf(D3X), **dma_buf**(D4X)

NAME

dma_get_cb – allocate a DMA command block

SYNOPSIS

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

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

ARGUMENTS

mode Specifies whether the caller is willing to sleep waiting for memory. If *mode* is set to **DMA_SLEEP**, the caller will sleep if necessary until the memory for a **dma_cb** is available. If *mode* is set to **DMA_NOSLEEP**, the caller will not sleep, but **dma_get_cb** will return **NULL** if memory for a **dma_cb** is not immediately available.

DESCRIPTION

dma_get_cb allocates memory for a DMA command block structure [see **dma_cb**(D4X)], zeroes it out, and returns a pointer to the structure.

RETURN VALUE

dma_get_cb returns a pointer to the allocated DMA control block. If **DMA_NOSLEEP** is specified and memory for a **dma_cb** is not immediately available, **dma_get_cb** returns a **NULL** pointer.

LEVEL

Base only if *mode* is set to **DMA_SLEEP**. Base or Interrupt if *mode* is set to **DMA_NOSLEEP**.

NOTES

Can sleep if *mode* is set to **DMA_SLEEP**.

SEE ALSO

dma_free_cb(D3X), **dma_cb**(D4X)

NAME

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

SYNOPSIS

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

void dma_stop(int chan);
```

ARGUMENTS

chan Channel on which DMA operation is to be stopped.

DESCRIPTION

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.

RETURN VALUE

None.

LEVEL

Base or Interrupt.

NOTES

Does not sleep.

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 currently in use for software-initiated DMA transfers rather than hardware-initiated transfers.

SEE ALSO

dma_swsetup(D3X), **dma_swstart**(D3X), **dma_cb**(D4X)

NAME

dma_swsetup – program a DMA operation for a subsequent software request

SYNOPSIS

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

int dma_swsetup(struct dma_cb *dmacbptr, int chan, uchar_t mode);
```

ARGUMENTS

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

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

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

DESCRIPTION

dma_swsetup programs the DMA channel *chan* for the operation specified by the DMA command block whose address is given by *dmacbptr*. Note that **dma_swsetup** does not initiate the DMA transfer. Instead, the transfer will be initiated by a subsequent request initiated via software by **dma_swstart**(D3X).

If **dma_swsetup** programs the operation successfully, it then calls the procedure specified by the **proc** field of the **dma_cb**(D4X) structure. It passes as an argument the value in the **procparms** field. If **proc** is set to **NULL**, then no routine is called.

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

RETURN VALUE

dma_swsetup returns the value **TRUE** on success and returns the value **FALSE** otherwise.

LEVEL

Base only if either (1) *mode* is set to **DMA_SLEEP** or (2) the routine specified by the **proc** field of the **dma_cb** structure sleeps. Base or Interrupt otherwise.

NOTES

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

SEE ALSO

dma_swstart(D3X), **dma_stop**(D3X), **dma_cb**(D4X)

NAME

buf – block I/O data transfer structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/page.h>
#include <sys/proc.h>
#include <sys/buf.h>
```

DESCRIPTION

The **buf** structure is the basic data structure for block I/O transfers. Each block I/O transfer has an associated buffer header. The header contains all the buffer control and status information. For drivers, the buffer header pointer is the sole argument to a block driver **strategy**(D2DK) 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**(D3DK)].

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

STRUCTURE MEMBERS

```
int             b_flags;        /* Buffer status */
struct buf      *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 */
uint_t          b_bcount;       /* # of bytes to transfer */
union {
      caddr_t   b_addr;         /* Buffer's virtual address */
} b_un;
daddr_t         b_blkno;        /* Block number on device */
uint_t          b_resid;        /* # of bytes not transferred */
clock_t         b_start;        /* Request start time */
struct proc     *b_proc;        /* Process structure address */
long            b_bufsize;      /* Size of allocated buffer */
int             (*b_iodone)();  /* Function called by biodone */
dev_t           b_edev;         /* Expanded dev field */
void            *b_private;     /* For driver's use */
```

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

b_resid indicates the number of bytes not transferred because of an error. The driver may change this member.

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 [see **vtop**(D3D)]. The driver may not change this member.

b_bufsize contains the size in bytes of the allocated buffer. The driver may not change this member unless the driver acquired the buffer with **getrbuf**.

(*b_iodone) identifies a specific driver routine to be called by the system when the I/O is complete. If one is specified, the **biodone**(D3DK) routine does not return the buffer to the system. The driver may change this member.

b_edev contains the external device number of the device.

b_private is a private field for use by the driver. The system does not interpret it. The driver is free to use it in whatever manner it chooses. For example, the driver could use it as part of a disk block sorting algorithm.

NOTES

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.

SEE ALSO

strategy(D2DK), **biodone**(D3DK), **bioerror**(D3DK), **biowait**(D3DK), **brelse**(D3DK), **clrbuf**(D3DK), **freerbuf**(D3DK), **geteblk**(D3DK), **geterror**(D3DK), **getrbuf**(D3DK), **ngeteblk**(D3DK), **physiock**(D3DK), **iovec**(D4DK), **uio**(D4DK)

SEE ALSO

Programmer's Guide: STREAMS

datab(D4DK), **msgb**(D4DK), **copyresp**(D4DK), **iocblk**(D4DK), **messages**(D5DK)

NAME

datab – STREAMS data block structure

SYNOPSIS

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

DESCRIPTION

The **datab** structure describes the data of a STREAMS message. The actual data contained in a STREAMS message is stored in a data buffer pointed to by this structure. A message block structure [**msgb**(D4DK)] 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 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.

NOTES

The **datab** structure is defined as type **dblk_t**.

SEE ALSO

Programmer's Guide: STREAMS

free_rtn(D4DK), **msgb**(D4DK), **messages**(D5DK)

NAME

iocblk – STREAMS **ioctl** structure

SYNOPSIS

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

DESCRIPTION

The **iocblk** structure describes a user's **ioctl**(2) request. It 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. When processing a transparent **ioctl**, the **iocblk** structure is usually overlaid with a **copyreq**(D4DK) structure. The stream head guarantees that the message is large enough to contain either structure.

STRUCTURE 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.

NOTES

Data cannot be copied to the user's buffer with an **M_IOCACK** message if the **ioctl** is transparent.

NAME

iovec – data storage structure for I/O using **uio**(D4DK)

SYNOPSIS

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

DESCRIPTION

An **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.

STRUCTURE 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**.

NOTES

A separate interface does not currently exist for allocating **iovec**(D4DK) structures when the driver needs to create them itself. Therefore, the driver may either use **kmem_zalloc**(D3DK) to allocate them, or allocate them statically.

SEE ALSO

physiock(D3DK), **uiomove**(D3DK), **ureadc**(D3DK), **uwritec**(D3DK), **uio**(D4DK)

NAME

module_info – STREAMS driver and module information structure

SYNOPSIS

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

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**(D4DK) structure, where they may be modified.

STRUCTURE 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.

NOTES

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.

NAME

msgb – STREAMS message block structure

SYNOPSIS

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

STRUCTURE 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**(D4DK). 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**(D4DK)] 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 queueing 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).

NOTES

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

SEE ALSO

Programmer's Guide: STREAMS
allocb(D3DK), **esballoc**(D3DK), **freeb**(D3DK), **datab**(D4DK), **free_rtn**(D4DK), **messages**(D5DK)

NAME

queue – STREAMS queue structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stream.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. 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 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**(D4DK) 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.

NAME

streamtab – STREAMS driver and module declaration structure

SYNOPSIS

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

DESCRIPTION

Each STREAMS driver or module must have a **streamtab** structure. The streamtab structure must be named *prefix***info**, where *prefix* is the driver prefix.

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.

STRUCTURE MEMBERS

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

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**.

SEE ALSO

qinit(D4DK)

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.

NAME

uio – scatter/gather I/O request structure

SYNOPSIS

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/uio.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**(D4DK) structures (base/length pairs) indicating where in user space or kernel space the data are to be read/written.

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**(D3DK), **ureadc**(D3DK), and **uwritec**(D3DK) functions take care of maintaining the the **uio** structure. A block driver may also use the **physiock**(D3DK) 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.

STRUCTURE 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 the driver is device-dependent. It applies to randomly accessed devices, but may not apply to all sequentially accessed devices.

NAME

dma_buf – DMA buffer descriptor structure

SYNOPSIS

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

DESCRIPTION

The DMA buffer descriptor structure is used to specify the data to be transferred by a DMA operation. Each DMA operation is controlled by a DMA command block [see **dma_cb**(D4X)] structure that includes pointers to two **dma_buf** structures.

Each **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 **dma_buf**s in a singly-linked list. Each **dma_buf** includes both the virtual and physical address of the next DMA buffer descriptor in the list.

DMA buffer descriptor structures should only be allocated via **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 MEMBERS

```
ushort_t        count;      /* size of block*/
paddr_t         address;    /* physical address of data block */
paddr_t         physical;   /* physical address of next dma_buf */
struct dma_buf  *next_buf;  /* next buffer descriptor */
ushort_t        count_hi;   /* for big blocks */
```

The members of the **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.

physical specifies the physical address of the next **dma_buf** in a linked list of DMA buffers descriptors. It should be **NULL** if the buffer descriptor is the last one in the list. Note that a DMA buffer descriptor allocated by **dma_get_buf** will be zeroed out initially, thus no explicit initialization is required for this field if a value of **NULL** is desired.

next_buf specifies the virtual address of the next **dma_buf** in a linked list of DMA buffer descriptors. It should be **NULL** if the buffer descriptor is the last one in the list. Note that a DMA buffer descriptor allocated by **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 **dma_buf** allocated by **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**.

NAME

dma_cb – DMA command block structure

SYNOPSIS

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

DESCRIPTION

The DMA command block structure is used to control a DMA operation. Each DMA operation requested by a driver is controlled by a command block structure whose fields specify the operation to occur.

A number of fields of the DMA control block come in pairs: one for the requestor and one for the 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.

DMA command block structures should only be allocated via **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 MEMBERS

```
struct dma_buf  *targbufs;   /* list of target data buffers */
struct dma_buf  *reqrbufs;   /* list of requestor data buffers */
uchar_t         command;     /* Read/Write/Translate/Verify */
uchar_t         targ_type;   /* Memory/IO */
uchar_t         reqr_type;   /* Memory/IO  */
uchar_t         targ_step;   /* Inc/Dec/Hold  */
uchar_t         reqr_step;   /* Inc/Dec/Hold  */
uchar_t         trans_type;  /* Single/Demand/Block/Cascade */
uchar_t         targ_path;   /* 8/16/32 */
uchar_t         reqr_path;   /* 8/16/32 */
uchar_t         cycles;      /* 1 or 2 */
uchar_t         bufprocess;  /* Single/Chain/Auto-Init */
char            *procparam;  /* parameter buffer for appl call */
int             (*proc)();   /* address of application call routines */
```

The members of the **dma_cb** structure are:

targbufs is a pointer to a list of DMA buffer structures [see **dma_buf**(D4X)] that describes the target of the DMA operation.

reqrbufs is a pointer to a list of DMA buffer structures [see **dma_buf**(D4X)] that describes the requestor of the DMA operation.

command specifies the command for the DMA operation. It may be one of the following:

DMA_CMD_READ	Specifies a DMA read from the target to the requestor.
DMA_CMD_WRITE	Specifies a DMA write from the requestor to the target.

SEE ALSO

dma_free_cb(D3X), dma_get_best_mode(D3X), dma_get_cb(D3X), dma_prog(D3X), dma_swsetup(D3X), dma_swstart(D3X), dma_buf(D4X)

NAME

errnos – error numbers

SYNOPSIS

```
#include <sys/errno.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**(D3DK) or **allocb**(D3DK).

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.

EFAULT Bad address. Drivers should return this error whenever a call to **copyin**(D3DK) or **copyout**(D3DK) 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.

NAME

messages – STREAMS messages

SYNOPSIS

```
#include <sys/stream.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.

NAME

signals – signal numbers

SYNOPSIS

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

DESCRIPTION

There are two ways to send a signal to a process. The first, **proc_signal**(D3DK), 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.
SIGWINCH	The window size has changed.
SIGURG	Urgent data are available.
SIGPOLL	A pollable event has occurred.
SIGTSTP	Interactive stop of the process.

NOTES

The signal **SIGTSTP** cannot be generated with **proc_signal**. It is only valid when generated from a stream.

SEE ALSO

proc_ref(D3DK), **proc_signal**(D3DK), **proc_unref**(D3DK)

- Replaced. The BCI routine has been removed from the DDI/DKI. The DDI/DKI provides a new interface that provides a similar function.
- Obsolete interface. The BCI routine has been removed from the DDI/DKI. The DDI/DKI does not provide a new interface; the interface itself is obsolete. For instance, the DDI/DKI does not support clist-based drivers; thus any routines dealing with clists have been removed from the DDI/DKI.

Again, please note that this table is a guide for programmers attempting to convert old driver source from BCI to DDI/DKI.

Table A-1: 3.2 to Release 4 Multi-Processor Migration

BCI	Comments	Release 4 Multi-Processor DDI/DKI
`adjmsg`	No change	`adjmsg`
`allocb`	No change; for memory-mapped I/O, use `esballoc`	`allocb`
`backq`	Obsolete interface.	—
`bcopy`	No change	`bcopy`
`brelse`	No change	`brelse`
`btoc`	Replaced	`btop`, `btopr`
`bufcall`	No change; don't use with `esballoc`	`bufcall`
`bzero`	Word alignment no longer required	`bzero`
`canon`	Obsolete interface.	—
`canput`	New restrictions; use `canputnext(q)` instead of `canput(q->q_next)`; stream cannot be frozen; use `bcanput` to test specific priority band	`canput`
`clrbuf`	No change	`clrbuf`
`cmn_err`	New restrictions; cannot hold locks if *level* is `CE_PANIC`	`cmn_err`
`copyb`	No change	`copyb`
`copyin`	New restrictions; cannot hold basic locks or read/write locks	`copyin` or `uiomove`
`copymsg`	No change	`copymsg`

Table A-1: 3.2 to Release 4 Multi-Processor Migration (continued)

BCI	Comments	Release 4 Multi-Processor DDI/DKI
	read/write locks; use **ngeteblk** or **getrbuf** for alternate buffer sizes	
getq	New restrictions; stream cannot be frozen	**getq**
inb	No change	**inb**
ind	Renamed only	**inl**
insq	New restrictions; stream must be frozen	**insq**
inw	No change	**inw**
iodone	Renamed only	**biodone**
iomove	Replaced	**uiomove**
iowait	Renamed and new restrictions; cannot hold basic locks or read/write locks	**biowait**
kseg	Obsolete interface.	**kmem_alloc**
linkb	No change	**linkb**
longjmp	Obsolete interface.	—
major	Renamed; macro reimplemented as function	**getmajor**
makedev	Renamed; macro reimplemented as function	**makedevice**
malloc	Renamed only	**rmalloc**
mapinit	Replaced	**rmallocmap**
mapwant	Replaced	**rmalloc_wait**
max	No change	**max**
mfree	Renamed only	**rmfree**
min	No change	**min**
minor	Renamed; macro reimplemented as function	**getminor**
msgdsize	No change	**msgdsize**
noenable	Macro reimplemented as function and new restrictions; stream cannot be frozen	**noenable**
OTHERQ	Macro reimplemented as function	**OTHERQ**
outb	No change	**outb**
outd	Renamed only	**outl**
outw	No change	**outw**
physck	Replaced; functionality included in **physiock**	**physiock**

Table A-1: 3.2 to Release 4 Multi-Processor Migration (continued)

BCI	Comments	Release 4 Multi-Processor DDI/DKI
`rmvb`	No change	`rmvb`
`rmvq`	New restrictions; stream must be frozen	`rmvq`
`signal`	Obsolete interface.	—
`sleep`	Replaced	`SV_WAIT_SIG`
`spl`	Replaced; `spl0, spl1, spl4, spl5, spl6, spl7` functions eliminated; `splbase, spltimeout, spldisk` added	`spl`
`splx`	No change	`splx`
`sptalloc`	Obsolete interface.	`kmem_alloc` or `physmap`
`sptfree`	Obsolete interface.	`kmem_free`
`strlog`	No change	`strlog`
`subyte`	Replaced	`copyout`, `uiomove`, or `ureadc`
`suser`	Replaced	`drv_priv`
`suword`	Replaced	`copyout`, `uiomove`, or `ureadc`
`testb`	Obsolete interface.	—
`timeout`	Replaced	`itimeout`
`ttclose`	Obsolete interface.	—
`ttin`	Obsolete interface.	—
`ttinit`	Obsolete interface.	—
`ttiocom`	Obsolete interface.	—
`ttioctl`	Obsolete interface.	—
`ttopen`	Obsolete interface.	—
`ttout`	Obsolete interface.	—
`ttread`	Obsolete interface.	—
`ttrstrt`	Obsolete interface.	—
`tttimeo`	Obsolete interface.	—

convert old driver source from Release 4 to Release 4 Multi-Processor. All routines in the Release 4 DDI/DKI, regardless of their status in the Release 4 Multi-Processor DDI/DKI, are provided in System V Release 4 Multi-Processor for Intel Processors for compatibility.

Table B-1: Release 4 to Release 4 Multi-Processor Migration

Release 4 DDI/DKI	Comments	Release 4 Multi-Processor DDI/DKI
`bcanput`	New restrictions; use `bcanputnext(q, pri)` instead of `bcanput(q->q_next, pri)`; stream cannot be frozen	`bcanput`
`biowait`	New restrictions; cannot hold basic locks or read/write locks	`biowait`
`bp_mapin`	New restrictions; cannot hold basic locks or read/write locks	`bp_mapin`
`canput`	New restrictions; use `canputnext(q)` instead of `canput(q->q_next)`; stream cannot be frozen	`canput`
`chpoll`	New restrictions; size of `pollhead` structure is not guaranteed; may not call any function that sleeps	`chpoll`
`cmn_err`	New restrictions; cannot hold locks if *level* is `CE_PANIC`	`cmn_err`
`copyin`	New restrictions; cannot hold basic locks or read/write locks	`copyin`
`copyout`	New restrictions; cannot hold basic locks or read/write locks	`copyout`
`delay`	New restrictions; cannot hold basic locks or read/write locks	`delay`
`dma_pageio`	New restrictions; cannot hold basic locks or read/write locks	`dma_pageio`
`enableok`	New restrictions; stream cannot be frozen	`enableok`
`flushband`	New restrictions; stream cannot be frozen	`flushband`
`flushq`	New restrictions; stream cannot be frozen	`flushq`

Table B-1: Release 4 to Release 4 Multi-Processor Migration (continued)

Release 4 DDI/DKI	Comments	Release 4 Multi-Processor DDI/DKI
`putnext`	New restrictions; cannot hold locks; stream cannot be frozen	`putnext`
`putq`	New restrictions; stream cannot be frozen	`putq`
`qenable`	New restrictions; stream cannot be frozen	`qenable`
`qreply`	New restrictions; cannot hold locks; stream cannot be frozen	`qreply`
`qsize`	New restrictions; stream cannot be frozen	`qsize`
`RD`	Extended. Accepts both read and write queue pointers	`RD`
`rminit`	Replaced	`rmallocmap`
`rmsetwant`	Replaced	`rmalloc_wait`
`rmvq`	New restrictions; stream must be frozen	`rmvq`
`SAMESTR`	New restrictions; argument cannot reference `q_next`; stream cannot be frozen	`SAMESTR`
`sleep`	Replaced	`SV_WAIT_SIG`
`spl`	Replaced; `spl0, spl1, spl4, spl5, spl6, spl7` functions eliminated; `splbase, spltimeout, spldisk` added	`spl`
`strqget`	New restrictions; stream must be frozen	`strqget`
`strqset`	New restrictions; stream must be frozen	`strqset`
`timeout`	Replaced	`itimeout`
`uiomove`	New restrictions; cannot hold basic locks or read/write locks if *uio_segflg* is `UIO_USERSPACE`	`uiomove`
`unbufcall`	Interface changed and new restrictions; argument type changed from `int` to `toid_t`; cannot hold locks	`unbufcall`
`untimeout`	Interface changed and new restrictions; argument type changed from `int` to `toid_t`; cannot hold locks	`untimeout`

Table B-2: Additions to the DDI/DKI in Release 4 Multi-Processor (continued)

Routine	Section	Description
RW_DEALLOC	D3DK	deallocate an instance of a read/write lock
RW_RDLOCK	D3DK	acquire a read/write lock in read mode
RW_TRYRDLOCK	D3DK	try to acquire a read/write lock in read mode
RW_TRYWRLOCK	D3DK	try to acquire a read/write lock in write mode
RW_UNLOCK	D3DK	release a read/write lock
RW_WRLOCK	D3DK	acquire a read/write lock in write mode
SLEEP_ALLOC	D3DK	allocate and initialize a sleep lock
SLEEP_DEALLOC	D3DK	deallocate an instance of a sleep lock
SLEEP_LOCK	D3DK	acquire a sleep lock
SLEEP_LOCKAVAIL	D3DK	query whether a sleep lock is available
SLEEP_LOCKOWNED	D3DK	query whether a sleep lock is held by the caller
SLEEP_LOCK_SIG	D3DK	acquire a sleep lock
SLEEP_TRYLOCK	D3DK	try to acquire a sleep lock
SLEEP_UNLOCK	D3DK	release a sleep lock
SV_ALLOC	D3DK	allocate and initialize a synchronization variable
SV_BROADCAST	D3DK	wake up all processes sleeping on a synchronization variable
SV_DEALLOC	D3DK	deallocate an instance of a synchronization variable
SV_SIGNAL	D3DK	wake up one process sleeping on a synchronization variable
SV_WAIT	D3DK	sleep on a synchronization variable
SV_WAIT_SIG	D3DK	sleep on a synchronization variable
TRYLOCK	D3DK	try to acquire a basic lock
UNLOCK	D3DK	release a basic lock
bcanputnext	D3DK	test for flow control in a specified priority band
bioerror	D3DK	manipulate error field within a buffer header
canputnext	D3DK	test for flow control in a stream
dtimeout	D3DK	execute a function on a specified processor,

NAME

ics_find _rec – reads the interconnect register of the board in the specified slot.

SYNOPSIS

```
#include <sys/ics.h>
int ics_find_rec (slot, recordid)
unsigned short slot;
unsigned char recordid;
```

ARGUMENTS

slot	the slot number of the board that will be searched
recordid	the record ID of the searched-for record

DESCRIPTION

ics_find_rec finds a specific record in the interconnect space of a board.

RETURN VALUE

If the searched-for record is found, its starting register number is returned. Otherwise, -1 is returned.

LEVEL

Base or Interrupt

SEE ALSO

ics_read(D3D), **ics_write**(D3D)

NAME

ics_rdwr – reads or writes a specified number of interconnect space registers from a given cardslot ID

SYNOPSIS

```
#include <sys/ics.h>
void ics_rdwr (cmd, addr)
int cmd;
struct ics_rw_struct *addr;
```

ARGUMENTS

cmd	Either **ICS_READ_ICS** or **ICS_WRITE_ICS**.
addr	A pointer to the description of the buffers to be used for the transfer.

DESCRIPTION

The **ics_rdwr** routine reads or writes a specified number of interconnect space registers from a given cardslot ID.

In both interconnect space and in memory, *addr* is a pointer to the description of the buffers to be used for the transfer. *addr* contains fields for length and addresses.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

ics_read(D3D), **ics_write**(D3D)

NAME

ics_write – writes a value into the specified register of the board in the specified slot.

SYNOPSIS

```
#include <sys/ics.h>
int ics_write (slot, register, value)
unsigned short slot;
unsigned short register;
unsigned char value;
```

ARGUMENTS

slot	The slot id of the board.
register	The register number of the board's interconnect space record.
value	The value to be written into the specified register

DESCRIPTION

ics_write writes *value* into register number register of the board in slot number slot. If no board is in the designated slot, the results are undefined.

RETURN VALUE

If the write is successful, 0 is returned. If the register number specified does not exist in the interconnect space of the board, **EINVAL** is returned.

LEVEL

Base or Interrupt

SEE ALSO

ics_read(D3D), **ics_rdwr**(D3D)

NAME

mps_AMPreceive_frag – receives solicited data in fragments when buffer space is not available at the receiving agent

SYNOPSIS

```
#include <sys/mps.h>
long mps_AMPreceive_frag(chan, mbp, socid, tid, ibuf)
long chan;
mps_msgbuf_t *mbp;
mb2socid_t socid;
unsigned char tid;
struct dma_buf *ibuf;
```

ARGUMENTS

chan Channel number received from a previous **mps_open_chan**.

mbp Points to a message buffer.

socid Identifies socket id of the socket which initiated the transaction.

tid Identifies the transaction corresponding to this **mps_AMPreceive_frag**. It is obtained from the request message.

ibuf Specifies the data buffer to receive incoming data. Indication of completion of transfer is sent to *intr* via a message.

DESCRIPTION

mps_AMPreceive_frag is used when an agent sending solicited data requests buffer space that is not available at the receiving agent. After the Buffer Reject message is sent, the receiving agent can use **mps_AMPreceive_frag** to receive the solicited data in fragments depending on the available buffer space in the receiving agent. See the *Multibus II Transport Protocol Specification and Designer's Guide* for additional information.

The **mps_AMPreceive_frag** routine queues up the message to initiate the transfer, sets up table entries to receive data messages, and returns immediately. This routine is asynchronous in operation.

Applications must ensure that **mps_AMPreceive_frag** is repeatedly used the correct number of times with the correct fragment buffer length to transfer an entire request.

RETURN VALUE

If no error is detected, 0 (zero) is returned. When an error is detected, −1 is returned.

LEVEL

Base or Interrupt

SEE ALSO

mps_open_chan(D3D)

NAME

mps_AMPsend_rsvp – queues request messages for transmission and sets up table entries for reply messages

SYNOPSIS

```
#include <sys/mps.h>
long mps_AMPsend_rsvp(chan, omsg, obuf, ibuf)
long chan;
mps_msgbuf_t *omsg;
struct dma_buf *obuf, *ibuf;
```

ARGUMENTS

chan	Channel number received from a previous **mps_open_ chan**.
omsg	Points to a message buffer containing message to be sent.
obuf	Specifies a data buffer for data to be sent.
ibuf	Specifies a data buffer to receive replies.

DESCRIPTION

mps_AMPsend_rsvp queues up request messages for transmission and sets up table entries for reception of reply messages when they arrive. This routine is asynchronous in operation.

When *obuf* is NULL, the request message is assumed to be an unsolicited message. In this case **mps_mk_unsol** (with a non-zero *tid* obtained by a call to **mps_get_tid**) should be used to build the message in *omsg*. When *obuf* is not NULL, request message is assumed to be a solicited message and *obuf* points to the data. In this case **mps_mk_unsol** (with a non-zero *tid* obtained by a call to **mps_get_tid**) should be used to build the message in *omsg*.

When *obuf* is not NULL, the request message is assumed to be a solicited message and *obuf* points to the solicited data. In this case, **mps_mk_sol** (with a non-zero tid obtained by a call to **mps_get_tid**) should be used to build the message in *omsg*. If *ibuf* is NULL, the reply message is expected to be an unsolicited message.

RETURN VALUE

mps_AMPsend_rsvp returns 0 (zero) if no error is detected; otherwise, –1 is returned.

LEVEL

Base or Interrupt

SEE ALSO

mps_open_chan(D3D), **mps_mk_sol**(D3D), **mps_mk_unsol**(D3D), **mps_get_tid**(D3D)

NAME

mps_AMPsend_reply – replies to a received request that is part of a request-response transaction

SYNOPSIS

```
#include <sys/mps.h>
long mps_AMPsend_reply(chan, omsg, obuf)
long chan;
mps_msgbuf_t *omsg;
struct dma_buf *obuf;
```

ARGUMENTS

chan Channel number received from a previous **mps_open_chan**.

omsg Points to a message buffer containing the message to be sent. The message in *omsg* should be constructed using **mps_mk_solrply** or **mps_mk_unsolrply** (depending on whether *obuf* is NULL or not) with the EOT flag set appropriately.

obuf Points to a data buffer containing data to be sent. When *obuf* is NULL, the reply message is assumed to be an unsolicited message. When *obuf* is not NULL, the reply message is assumed to be a solicited message. A completion indication is sent via a message to the appropriate **intr** routine.

DESCRIPTION

mps_AMPsend_reply is used to send a reply in response to a received request that is part of a request-response transaction. The **mps_AMPsend_reply** routine is asynchronous in operation. **mps_AMPsend_reply** returns immediately, queuing up to send the reply. Be sure to use the *tid* from the corresponding received request.

mps_AMPsend_reply can be used to send a reply as a number of solicited fragments. The message buffer in the last reply fragment should have the *EOT* flag set to 1.

RETURN VALUE

If no error is detected, 0 (zero) is returned; otherwise –1 is returned.

LEVEL

Base or Interrupt

SEE ALSO

mps_mk_solrply(D3D), **mps_mk_unsolrply**(D3D), **mps_open_chan**(D3D)

NAME

mps_close_chan – closes a previously opened channel

SYNOPSIS

```
#include <sys/mps.h>
long mps_close_chan (chan)
long chan;
```

ARGUMENTS

chan Specifies the channel to be closed.

DESCRIPTION

This routine is used to close a previously opened channel. To close a channel a device driver must identify the channel.

The **mps_close_chan** routine is synchronous in operation. **mps_close_chan** fails if a transaction is in progress on the specified channel.

RETURN VALUE

When **mps_close_chan** succeeds it returns 0 (zero). When **mps_close_chan** fails, it returns –1 and the channel is not closed.

LEVEL

Base or Interrupt

SEE ALSO

mps_open_chan(D3D)

NAME

mps_free_msgbuf – puts a buffer back into the free memory pool

SYNOPSIS

```
#include <sys/mps.h>
void mps_free_msgbuf(mbp)
mps_msgbuf_t *mbp;
```

ARGUMENTS

mbp the message buffer to be returned to the free memory pool.

DESCRIPTION

In this function, *mbp* points to a message buffer. The buffer is put back in the free memory pool. Note that **mps_free_msgbuf** accepts a pointer to a single message buffer, not a list of message buffers to be freed.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

mps_get_msgbuf(D3D)

NAME

mps_get_dmabuf – returns a pointer to a list of data buffer descriptors.

SYNOPSIS

```
#include <sys/mps.h>
struct dma_buf *mps_get_dmabuf(count, flag)
unsigned int    count;
int flag;
```

ARGUMENTS

count the number of dma buffer descriptors required.

flag determines whether the routine sleeps while waiting for resources. Valid values are **DMA_SLEEP** or **DMA_NOSLEEP**.

DESCRIPTION

The **mps_get_dmabuf** function returns a pointer to a linked list of (*count* number of) data buffer descriptors. The list is terminated by NULL in the *db_next* field of the data buffer.

RETURN VALUE

If *count* number of data buffer descriptors cannot be allocated, and *flag* = **DMA_NOSLEEP**, a NULL descriptor is returned. Otherwise, if *flag* = **DMA_SLEEP**, the routine blocks until *count* data buffer descriptors can be allocated.

LEVEL

Base or Interrupt with DMA_NOSLEEP

SEE ALSO

mps_free_dmabuf(D3D)

NAME

mps_get_reply_len – get data length for a solicited reply.

SYNOPSIS

```
#include <sys/mps.h>
long mps_get_reply_len(socid, tid)
mb2socid_t      socid;
unsigned char   tid;
```

ARGUMENTS

socid	The source socid for the solicited reply
tid	the transaction id of the solicited reply

DESCRIPTION

This function should be invoked when an rsvp completes with an unsolicited message, instead of with a a solicited message; that is, when the flags field of the final message buffer is MPS_MG_UNSOL. In this case, the **mps_get_reply_len** function returns the length of the data for the solicited reply associated with the rsvp when it is called after the transaction completes.

RETURN VALUE

A successful operation returns the length of the data. If an error occurs, 0 is returned as the data length.

LEVEL

Base or Interrupt

NAME

mps_get_tid – allocates transaction ids.

SYNOPSIS

```
#include <sys/mps.h>
unsigned char mps_get_tid(chan)
long chan;
```

ARGUMENTS

chan a channel number obtained from a previous call to **mps_open_chan**.

DESCRIPTION

The **mps_get_tid** function is used by users of the message handler to allocate transaction ids.

RETURN VALUE

If no free transaction ids are available for the associated port id, or when *chan* is an invalid channel number, 0 (zero) is returned; otherwise the allocated transaction id is returned.

LEVEL

Base or Interrupt

SEE ALSO

mps_open_chan(D3D), **mps_free_tid**(D3D)

NAME

mps_mk_bgrant – construct a buffer grant in response to a buffer request.

SYNOPSIS

```
#include <sys/mps.h>
void mps_mk_bgrant(mbp, dsocid, lid, count)
mps_msgbuf_t          mbp;
mb2socid_t            dscocid;
unsigned char         lid;
unsigned long         count;
```

ARGUMENTS

mbp	pointer to message buffer
dsocid	32-bit destination socket id (host id:port id)
lid	liaison id
count	number of bytes to transfer

DESCRIPTION

The **mps_mk_bgrant** function is used to construct a buffer grant in response to a buffer request. Arguments to this function are not checked for valid values.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

mps_mk_unsolrply(D3D)

NAME

mps_mk_breject – construct a buffer reject in response to a buffer request.

SYNOPSIS

```
#include <sys/mps.h>
void mps_mk_breject(mbp, dsocid, lid)
mps_msgbuf_t          mbp;
mb2socid_t            dscocid;
unsigned char         lid;
```

ARGUMENTS

mbp	pointer to message buffer
dsocid	32-bit destination socket id (host id:port id)
lid	liaison id

DESCRIPTION

The **mps_mk_breject** function is used to construct a buffer reject in response to a buffer request. Arguments to this function are not checked for valid values.

RETURN VALUE

None

LEVEL

Base or Interrupt

NAME

mps_mk_solrply – constructs a message to be sent to initiate a solicited data reply.

SYNOPSIS

```
#include <sys/mps.h>
void mps_mk_solrply(mbp, dsocid, tid, dptr, count, eotflag)
mps_msgbuf_t          mbp;
mb2socid_t            dscocid;
unsigned char         tid;
unsigned char         *dptr;
unsigned long         count;
unsigned char         eotflag;
```

ARGUMENTS

mbp	pointer to message buffer
dsocid	32-bit destination socket id (host id:port id)
tid	8-bit transaction id
dptr	pointer to user data to be sent with the message
count	number of bytes of user data to be sent with the message (Max 16)
eotflag	1 to indicate end of transaction; otherwise, 0 (zero)

DESCRIPTION

The **mps_mk_solrply** function takes a pointer to a message buffer and constructs a message to be sent to initiate a solicited data reply. The message is constructed using values supplied as arguments. Arguments to this function are not checked for valid values.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

mps_mk_unsolrply(D3D)

NAME

mps_mk_unsolrply – constructs a unsolicited reply message to be sent.

SYNOPSIS

```
#include <sys/mps.h>
void mps_mk_unsolrply(mbp, dsocid, tid, dptr, count)
mps_msgbuf_t          mbp;
mb2socid_t            dscocid;
unsigned char         tid;
unsigned char         *dptr;
unsigned long         count;
```

ARGUMENTS

mbp	pointer to message buffer
dsocid	32-bit destination socket id (host id:port id)
tid	8-bit transaction id
dptr	pointer to user data to be sent with the message
count	number of bytes of user data to be sent with the message (Max 20)

DESCRIPTION

The **mps_mk_unsolrply** function takes a pointer to a message buffer and constructs a unsolicited reply message to be sent. The message is constructed using values supplied as arguments. Arguments to this function are not checked for valid values.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

mps_mk_solrply(D3D)

RETURN VALUE

Listed above.

LEVEL

Base or Interrupt

UNIX® SYSTEM V RELEASE 4

Device Driver Interface / Driver-Kernel Interface Reference Manual

Intel Processors

The reference manual set for UNIX® System V Release 4 for Intel Processors is the definitive source for complete and detailed specifications for all System V interfaces. Newly reorganized, this edition makes finding the manual page you need easy and fast.

The new organization groups manual pages in the way most users need to use them:

- The *User's Reference Manual/System Administrator's Reference Manual* describes all user and administrator commands in the UNIX system, including new multiprocessing commands.
- The *Programmer's Reference Manual:Operating System API* describes UNIX system calls and C language library functions, including new multiprocessing interfaces.
- The *System Files and Devices Reference Manual* describes file formats, special files (devices), and miscellaneous system facilities.
- The *Device Driver Interface/Driver-Kernel Interface Reference Manual* describes functions used by device driver software. Editions of this manual are available for both uniprocessor and multiprocessor versions of the operating system.
- The *Product Overview and Master Index* provides an overview of the system and comprehensive indices for the documentation set.

Use Background Color to Locate Your Document Title:

COLOR CODE	DOCUMENT TYPE
	GENERAL DOCUMENTS
	USER'S GUIDES
	ADMINISTRATOR'S GUIDES
	PROGRAMMER'S GUIDES
	REFERENCE MANUALS

UNIX PRESS
A Prentice Hall Title

ISBN 0-13-879529-0
9 780138 795290 90000>