Legal Notices

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaption, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-703.

Copyright ©, 1988, The Santa Cruz Operation

Copyright ©, 1979, 1980, 1983, The Regents of the University of California

Copyright ©, 1980, 1984 AT&T, Inc

Copyright ©, 1986, 1992 Sun Microsystems, Inc


This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.
Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections that are incorporated at a reprinting do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

**Edition / Part Number / Date**

First Edition / B2355-90670 / September 1999

Second Edition / B2355-90726 / August 2000
Contents

1. Introduction
   The Intended Audience .................................................. 13
   Support/Compatibility Disclaimers ............................. 14
   Reference Page Format ............................................. 15
   Reference Page Sections ........................................... 16

2. Kernel Reference Pages
   Functions and Structures .......................................... 19
   alloc_spinlock(KER2) .................................................. 20
   b_cpsema(KER2) ....................................................... 22
   b_initsema(KER2) ..................................................... 25
   b_owns_sema(KER2) ................................................... 27
   b_psema(KER2) ......................................................... 29
   b_vsema(KER2) ........................................................ 31
   bcmp(KER2) ......................................................... 33
   bcopy(KER2) ........................................................ 34
   biodone(KER2) ......................................................... 36
   biowait(KER2) ......................................................... 38
   brelse(KER2) ........................................................ 40
   buf(KER4) .......................................................... 41
   busywait(KER2) ......................................................... 46
   bzero(KER2) .......................................................... 47
   copyin(KER2) ........................................................ 48
   copyout(KER2) ........................................................ 50
   cspinlock(KER2) ....................................................... 52
   dealloc_spinlock(KER2) ........................................... 55
   FREE(KER2) Kernel Macro ......................................... 56
   get_sleep_lock(KER2) .............................................. 58
   getc(KER2) .......................................................... 60
   getcb(KER2) .......................................................... 62
   getcf(KER2) .......................................................... 64
   geteblk(KER2) ......................................................... 65
   geterror(KER2) ........................................................ 67
   gsignal(KER2) ......................................................... 69
   iovec(KER4) .......................................................... 71
   isc_table_type(KER4) ............................................... 72
kernel_iomap(KER2) ................................................................. 77
kernel_iomap_public(KER2) .................................................. 80
kernel_iounmap(KER2) ......................................................... 83
kfree(KER2) ........................................................................... 84
kmalloc(KER2) ................................................................. 85
Ktimeout(KER2) ................................................................. 87
lstat(KER2) ................................................................. 89
major(KER2) ........................................................................ 91
makedev(KER2) ............................................................... 92
MALLOC(KER2) ............................................................... 93
map_mem_to_host(KER2) ................................................. 96
minor(KER2) ................................................................. 98
minphys(KER2) ............................................................ 99
msg_printf(KER2) .......................................................... 101
owns_spinlock(KER2) ..................................................... 103
p_pgrp(KER2) ......................................................................... 105
panic(KER2) ......................................................................... 107
physio(KER2) ................................................................. 108
printf(KER2) ................................................................. 112
privlbcopy(KER2) ........................................................... 114
psignal(KER2) ............................................................... 116
putc(KER2) ........................................................................... 118
putcb(KER2) ......................................................................... 120
putcf(KER2) .......................................................................... 121
selwakeup(KER2) ........................................................... 123
sleep(KER2) ................................................................. 125
spinlock(KER2) ............................................................. 128
spinunlock(KER2) .......................................................... 130
sprintf(KER2) ............................................................... 132
strcat(KER2) ................................................................. 134
strcmp(KER2) ................................................................. 135
strlen(KER2) ......................................................................... 137
strncpy(KER2) ............................................................... 139
strncmp(KER2) .............................................................. 141
strncpy(KER2) ............................................................... 143
suser(KER2) ................................................................. 145
sw_trigger(KER2) .......................................................... 146
timeout(KER2) ............................................................... 149
uio(KER2) ................................................................. 151
uniomove(KER2) ........................................................... 153
unmap_mem_from_host(KER2) ........................................... 156
untimeout(KER2) ........................................................... 157
3. CDIO Reference Pages
Structure and Macro Commands ........................................ 167
  dma_sync(CDIO3) ......................................................... 168
  dma_sync_IO(CDIO3) ....................................................... 171
  drv_info(CDIO4) .......................................................... 174
  drv_ops(CDIO4) ............................................................ 177
  init_map_context(CDIO3) ............................................... 181
  install_driver(CDIO3) ................................................... 183

4. WSIO Reference Pages
Functions, Macros and Structures ........................................ 187
  dma_cleanup(WSIO3) ....................................................... 188
  dma_parms(CDIO4) ........................................................ 189
  dma_setup(WSIO3) ........................................................ 191
  driver_addr_probe(WSIO_DRV) .......................................... 192
  driver_attach(WSIO_DRV) ................................................. 194
  driver_class_probe(WSIO_DRV) ......................................... 196
  driver_close(WSIO_DRV) ............................................... 198
  driver_dev_init(WSIO_DRV) ............................................. 200
  driver_if_init(WSIO_DRV) ............................................... 201
  driver_install(WSIO_DRV) .............................................. 202
  driver_ioctl(WSIO_DRV) ............................................... 204
  driver_isr(WSIO_DRV) .................................................. 206
  driver_minor_build(WSIO_DRV) ....................................... 207
  driver_minphys(WSIO_DRV) ............................................. 209
  driver_open(WSIO_DRV) ................................................ 210
  driver_psize(WSIO_DRV) ............................................... 212
  driver_read(WSIO_DRV) ................................................ 213
  driver_select(WSIO_DRV) .............................................. 215
  driver_strategy(WSIO_DRV) ............................................ 217
  driver_write(WSIO_DRV) ................................................ 218
  free_isc(WSIO3) .......................................................... 220
  get_new_isc(WSIO3) ..................................................... 221
  iodone(WSIO3) ............................................................. 223
  iowait(WSIO3) ............................................................. 225
  isc_claim(WSIO3) ........................................................ 227
5. Network Device Driver Reference Pages

Network Macros, Structures, and Functions ........................................ 301
  ETHER_MAP_IP_MULTICAST ...................................................... 302
  hw_ift(NET4) Network Structure ............................................. 303
  hw_ift_attach(NET3) Network Function ....................................... 308
  HWIFT_LOCK(NET3), HWIFT_UNLOCK(NET3) Network Functions .......... 309
  kget_log_instance(NET3) Network Function .................................. 311
  KLOG_CK(NET3) Network Function ............................................ 312
  klogg_write(NET3) Network Function ......................................... 314
dd_ioctl(SCSI_DRV) SCSI Function ......................................................... 398
dd_ioctl_okay(SCSI_DRV) SCSI Function ........................................... 399
dd_open(SCSI_DRV) SCSI Function ...................................................... 400
dd_pass_thru_done(SCSI_DRV) SCSI Function ................................. 401
dd_pass_thru_okay(SCSI_DRV) SCSI Function ................................. 402
dd_read(SCSI_DRV) SCSI Function .................................................... 403
dd_start(SCSI_DRV) SCSI Function ................................................. 404
dd_strategy(SCSI_DRV) SCSI Function ............................................ 405
dd_write(SCSI3) SCSI Function ........................................................ 406
m_scsi_lun(SCSI13) SCSI Function ................................................... 407
scsi_action(SCSI3) SCSI Function .................................................... 408
scsi_cmd(SCSI3) SCSI Function ....................................................... 411
scsi_cmdx(SCSI3) SCSI Function ...................................................... 413
scsi_ddsw(SCSI4) SCSI Structure ...................................................... 415
scsi_dequeue(SCSI3) SCSI Function ................................................. 419
scsi_dequeue_bp(SCSI3) SCSI Function ............................................. 421
scsi_enqueue(SCSI3) SCSI Function .................................................. 422
scsi_init_inquiry_data(SCSI3) SCSI Function .................................... 423
scsi_ioctl(SCSI3) SCSI Function ........................................................ 424
scsi_lun_close(SCSI3) SCSI Function ................................................. 432
scsi_lun_open(SCSI) SCSI Function .................................................... 434
scsi_read(SCSI3) SCSI Function ........................................................ 436
scsi_sense_action(SCSI) SCSI Function ............................................. 437
scsi_strategy(SCSI3) SCSI Function .................................................. 439
scsi_write(SCSI3) SCSI Function ...................................................... 441
1 Introduction
Introduction

This reference manual contains manual reference pages and other reference information on the kernel support routines, data structures, services, and macros essential for developing HP-UX drivers. See HP-UX Driver Development Guide for further information on how to use these functions.
The Intended Audience

Porting an existing device driver is not a trivial task. Writing a device driver is even more complex. Using this manual to port or write a driver assumes that you know how to:

- Write programs in the C language.
- Understand the basic concepts of writing a driver.
- Understand the functionality of the hardware for which you are writing the driver.
- Read the HP-UX System Administration Tasks manual and perform system administration.
- In areas such as virtual memory, I/O, and file systems, understand the HP-UX and/or UNIX operating systems.

These assumptions are not meant to discourage anyone, but you should not plunge onward unless you know the HP-UX (UNIX) operating system, the C language, and the implications of writing drivers. The “Support/Compatibility Disclaimers” section describes the support provided by Hewlett-Packard Company.

NOTE

This book contains many examples of C programs to help you design device drivers. Because of page width restrictions, some long lines of code exceed the space available and break in unintended places. Please treat these “broken” lines as one line. We recommend that you use the sample files included with this manual, when possible, rather than retyping the examples.
Support/Compatibility Disclaimers

Since drivers function at the level of the kernel, Hewlett-Packard Company (HP) reminds you of the following things:

- Adding your own driver to HP-UX requires relinking the driver into HP-UX. With each new release you should plan on recompiling your driver in order to reinstall it into the new HP-UX kernel. Many header files do not change. However, drivers typically use some header files that could change across releases (i.e., you can have some system dependencies).

- The information in this manual is correct, to HP’s knowledge, but the information can change (e.g., kernel routines and header files).

- HP provides support services for HP products, including HP-UX. Products, including drivers, from non-HP parties receive no support, other than the support of those parts of a driver that rely on the documented behavior of supported HP products.

- Should difficulties arise during the development and test phases of writing a driver, HP may provide assistance in isolating problems to determine if:
  - HP hardware is not at fault; and
  - HP software (firmware) is not at fault by removing user-written kernel drivers.

- When HP hardware, software, and firmware are not at fault, you should seek help from the third party from whom you obtained software or hardware.
Reference Page Format

All manpage entries in this section follow an established topic format, but not all topics are included in each entry.

NAME  Gives the name of the entry and briefly states its purpose.

SYNOPSIS  Lists source code of the include file that defines the structure.

PARAMETERS  Defines the parameters of the routine.

DESCRIPTION  Provides general information about the structure, routine, or macro.

STRUCTURE MEMBERS  Lists all accessible structure members

RETURN VALUES  Describes the values the routine can return.

CONSTRAINTS  Identifies when a function can not be called.

WARNINGS  Provides suggestions to avoid potential problems or pitfalls that may result in lost time or data.

EXAMPLES  Gives sample program segments demonstrating the routine.

SEE ALSO  Provides pointers to related topics.
Reference Page Sections

Reference page references are in the form pagename(section), where pagename is the name of the page and section is the section name. Pages in this manual are assigned to the following section names, for example, bcopy(KER2), where the numbers correspond to those in HP-UX Reference.

CDIO3 A function available to drivers in any CDIO.
CDIO4 A structure used by CDIO function.
KER2 A kernel function available to all drivers.
KER4 A structure used by kernel functions.
NET3 A function available to networking drivers.
NET4 A structure used by networking functions.
NET_DRV A driver-supplied networking function.
PCI3 A function available in the PCI CDIO.
PCI5 PCI errata.
SCSI3 A function available in the SCSI CDIO.
SCSI_DRV A driver-supplied SCSI function.
WSIO3 A function available in the WSIO CDIO.
WSIO4 A structure used by WSIO functions.
WSIO_DRV A driver-supplied kernel function.

Reference pages in HP-UX Reference have one-digit section numbers, sometimes accompanied by a single letter, for example, open(2).
2 Kernel Reference Pages
Kernel Reference Pages

This chapter contains reference pages for the kernel support routines commonly used by I/O drivers.
Functions and Structures
NAME

alloc_spinlock(KER2) - Allocate and initialize a spinlock resource.

SYNOPSIS

#include <sys/spinlock.h>

lock_t *alloc_spinlock (unsigned int order, char *name);

PARAMETERS

order Lock order.
name Spinlock name.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The alloc_spinlock() kernel function allocates and initializes a spinlock resource. The caller is responsible for deallocating the spinlock resource when it is no longer needed. See dealloc_spinlock(KER2).

The order parameter is the lock order of the spinlock. The order must be chosen such that deadlocks with other spinlocks are avoided. To avoid deadlocks, the spinlock to be acquired must have a lock order greater than that of any spinlock currently held by the processor. See spinlock orders in <sys/semglobal.h>.

The name parameter points to a character string containing the name of the spinlock.

RETURN VALUES

alloc_spinlock() returns a pointer to a spinlock resource.
CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.

EXAMPLE

/*
 * Choose a lock order for my driver that is lower than
 * any spinlock used by a service that the driver may
 * call. In <sys/semglobal.h>, SPL_LOCK_ORDER is
 * defined with a sufficiently low lock order value.
 */
#define MYDRV_LOCK_ORDER SPL_LOCK_ORDER

lock_t * mydrv_lock;

/*
 * Allocate a spinlock resource. alloc_spinlock() does
 * not return until it has successfully allocated the
 * memory for the spinlock and initialized the spinlock.
 */
mydrv_lock = alloc_spinlock(MYDRV_LOCK_ORDER,
                           "mydrv spinlock");

SEE ALSO
cspinlock(KER2), dealloc_spinlock(KER2),
owns_spinlock(KER2), spinlock(KER2), spinunlock(KER2)
NAME

*b_cpsema* (KER2) - Conditionally acquire (lock) a beta semaphore.

SYNOPSIS

```c
#include <sys/sem_beta.h>

int b_cpsema (struct b_sema *sema);
```

PARAMETERS

*sema* Pointer to a *b_sema* structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The *b_cpsema()* kernel function attempts to conditionally acquire (lock) a beta semaphore pointed to by *sema*. The calling thread is not blocked if the beta semaphore is currently owned.

RETURN VALUES

*b_cpsema()* returns the following values:

- **1**: Acquired (locked) the beta semaphore
- **0**: The beta semaphore is currently owned.

CONSTRAINTS

- Must not be called in an interrupt context.
- Must not be called while holding a spinlock with lock order >= SEMAPHORE_LOCK_ORDER.
EXAMPLE

static b_sema_t mydrv_sema_1;
static b_sema_t mydrv_sema_2;
...

/*
 * Acquire a beta semaphore. This is the first of two
 * beta semaphores that will be needed, but we must
 * acquire mydrv_sema_2 before mydrv_sema_1 which is the
 * wrong lock order. We get away with doing this by
 * trying to conditionally acquire
 * mydrv_sema_1 later.
 */
b_psema(&mydrv_sema_2);
...

/*
 * Try to conditionally acquire another beta semaphore.
 * This violates the normal lock order (the previous
 * beta semaphore has a higher lock order), but this is
 * allowed since b_cpsema() does not block the thread.
 */
if (!b_cpsema(&mydrv_sema_1)) {
  /*
   * Failed to acquire the next beta semaphore.
   * As a sanity check, assert that we are not
   * the thread that owns the beta semaphore.
   */
  VASSERT(!b_owns_sema(&mydrv_sema_1));

  /*
   * Release mydrv_sema_2 to allow the other thread
   * that owns mydrv_sema_1 to make forward progress,
   * otherwise a deadlock condition will exist.
   */
  b_vsema(&mydrv_sema_2);

  /*
   * Now acquire the two beta semaphores in the correct
   * order to avoid a deadlock. We will probably block
   * here until the other thread releases mydrv_sema_1.
   */
  b_psema(&mydrv_sema_1);
  b_psema(&mydrv_sema_2);
/*
 * Check if data have changed between the time we released
 * mydrv_sema_2 and reacquired the beta semaphores.
 */

SEE ALSO

b_initsema(KER2), b_owns_sema(KER2), b_psema(KER2),
b_vsema(KER2)
NAME

*b_initsema*(KER2) – Initialize a beta semaphore.

SYNOPSIS

```c
#include <sys/sem_beta.h>

void b_initsema (struct b_sema *sema, int val, int order,
                  char *name);
```

PARAMETERS

- `sema` Pointer to a `b_sema` structure.
- `order` Lock order.
- `name` Beta semaphore name.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The `b_initsema()` kernel function initializes the beta semaphore pointed to by the `sema` parameter. The caller is responsible for allocating the kernel memory that instantiates the `b_sema` structure.

The `val` parameter is normally set to the value 1, indicating the beta semaphore is initially not owned (unlocked). If `val` is set to 0, the beta semaphore is initialized as owned (locked).

The `order` parameter is the lock order of the beta semaphore. `order` must be a positive value and chosen so that deadlocks with other beta semaphores are avoided. To avoid deadlocks, the semaphore to be acquired must have a lock order greater than that of any semaphore currently held by the thread. See beta-class semaphore lock orders in `<sys/semglobal.h>`.
If the `SEMA_DEADLOCK_SAFE` flag is set (ORed with `order`), deadlock detection is disabled for the special case where beta semaphores have the same lock order values. The caller takes responsibility for ensuring it acquires semaphores of equal lock order in a deadlock-safe manner. The caller, however, is not allowed to acquire a beta semaphore with a lock order less than that of another beta semaphore currently held by the thread.

The `name` parameter points to a character string containing the name of the beta semaphore. The character string must not be an empty string.

**RETURN VALUES**

None

**CONSTRAINTS**

Must not be called in an interrupt context.

Must not be called while holding a spinlock with lock order >= `LOCK_INIT_LOCK_ORDER`.

**EXAMPLE**

```c
/*
 * Allocate static storage for the beta semaphore.
 *
static b_sema_t mydrv_sema;
...

/*
 * Initialize the beta semaphore. The lock order value
 * REAL_DRV_SEMA_ORDER is defined in <sys/semglobal.h>.
 */
b_initsema(&mydrv_sema, 1, REAL_DRV_SEMA_ORDER,
          "mydrv sema");
```

**SEE ALSO**

`b_cpsema(KER2), b_owns_sema(KER2), b_psema(KER2), b_vsema(KER2)`
NAME

b_owns_sema (KER2) – Test whether a beta semaphore is owned by the calling thread.

SYNOPSIS

#include <sys/sem_beta.h>

int b_owns_sema (struct b_sema *sema);

PARAMETERS

sema Pointer to a b_sema structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The b_owns_sema() kernel function tests whether a beta semaphore, pointed to by sema, is owned (locked) by the calling thread.

RETURN VALUES

b_owns_sema() returns the following values

1 The calling thread owns the beta semaphore.

0 The current thread does not own the beta semaphore. It may be another thread, or not owned by any thread.

CONSTRAINTS

Must not be called in an interrupt context.
Kernel Reference Pages
Functions and Structures

**EXAMPLE**

```c
static b_sema_t mydrv_sema;
...

int got_sema_here = 0;

/*
 * We may have already acquired the semaphore before getting
 * here. Be sure to check that we do not own the semaphore
 * before attempting to acquiring the semaphore.
 */
if (!b_owns_sema(&mydrv_sema)) {
    b_psema(&mydrv_sema);
    got_sema_here = 1;
}

/*
 * Do work under the protection of the semaphore.
 */
...

/*
 * Release the semaphore if it was acquired here.
 */
if (got_sema_here) {
    b_vsema(&mydrv_sema);
}
```

**SEE ALSO**

`b_cpsema(KER2), b_initsema(KER2), b_psema(KER2), b_vsema(KER2)`
NAME

b_psema (KER2) – Acquire (lock) a beta semaphore.

SYNOPSIS

#include <sys/sem_beta.h>

int b_psema (struct b_sema *sema);

PARAMETERS

sema Pointer to a b_sema structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The b_psema() kernel function attempts to acquire (lock) a beta semaphore pointed to by sema. The calling thread is blocked if the beta semaphore is currently owned.

RETURN VALUES

None

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.

EXAMPLE

static b_sema_t mydrv_sema;
...
/*
 * Acquire the beta semaphore for my driver.
b_psema(&mydrv_sema);

/*
 * Manipulate driver data protected by the beta semaphore.
 * Note: It is OK to sleep while holding a beta semaphore.
 */
...

/*
 * Release the beta semaphore when done.
 */
b_vsema(&mydrv_sema);

SEE ALSO

b_cpsema(KER2), b_initsema(KER2), b_owns_sema(KER2), b_vsema(KER2)
NAME

b_vsema (KER2) – Release (unlock) a beta semaphore.

SYNOPSIS

#include <sys/sem_beta.h>

int b_vsema (struct b_sema *sema);

PARAMETERS

sema Pointer to a b_sema structure.

DESCRIPTION

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

The b_vsema() kernel function releases (unlocks) the beta semaphore pointed to by sema.

RETURN VALUES

None.

CONSTRAINTS

Must not be called in an interrupt context.

EXAMPLE

static b_sema_t mydrv_sema;
...

/*
 * Acquire the beta semaphore for my driver.
 */
 b_psema(&mydrv_sema);
/*
 * Manipulate driver data protected by the beta semaphore.
 * Note: It is OK to sleep while holding a beta semaphore.
 */
...

/*
 * Release the beta semaphore when done.
 */
b_vsema(&mydrv_sema);

SEE ALSO

b_cpsema(KER2), b_initsema(KER2), b_owns_sema(KER2),
b_psema(KER2),
NAME

bcmp(KER2) - Compare two byte arrays

SYNOPSIS

#include <sys/kern_svcs.h>

int bcmp (caddr_t s1, caddr_t s2, uint32_t n);

PARAMETERS

s1  Pointer to the first byte array.

s2  Pointer to the second byte array.

n   Number of bytes to compare.

DESCRIPTION

The bcmp() kernel function compares n bytes of the byte arrays starting at s1 and s2. If these n bytes are identical, return zero. If the n bytes are not identical, the integer value of (s1[k]-s2[k]) is returned, where k is the failing byte offset in the array. Unlike strcmp(), bcmp() does not terminate when it encounters a null byte.

RETURN VALUES

bcmp() returns the following values:

0   The byte arrays are identical.

<>0  The byte arrays are different.

CONSTRAINTS

SEE ALSO

strcmp(KER2)
NAME
bcopy(KER2) – Copy data from a source buffer to a destination buffer

SYNOPSIS
#include <sys/kern_svcs.h>

int bcopy (caddr_t from, caddr_t to, size_t n);

PARAMETERS
from Pointer to the source buffer.
to Pointer to the destination buffer.
n Number of bytes to copy.

DESCRIPTION
The bcopy() kernel function copies n bytes from a kernel space buffer to another kernel space buffer. The two buffers must not overlap.

To copy data between user space and kernel space, use copyin(), copyout(), or uiomove().
To copy data between buffers in other user spaces, use privbcopy().

RETURN VALUES
bcopy() returns the following values:
0 Successful completion.
-1 Error.

CONSTRAINTS

WARNINGS
The function bcopy() must not be used for transfers between memory and IO space. The underlying routines make choices of the optimal
transfer code which will probably not be supported on either the IO bus adapter or the attached interface card. Failure to heed this warning may result in data corruption, an HPMC (High Priority Machine Check), or a call to panic().

SEE ALSO

copyin(KER2), copyout(KER2), privlbcopy(KER2), uiomove(KER2)
NAME

biodone(KER2) – Complete the buffer I/O transaction.

SYNOPSIS

#include<sys/buf.h>

void biodone (struct buf * bp);

PARAMETERS

bp A pointer to a buf structure.

DESCRIPTION

The biodone() kernel function completes the buffer I/O transaction. There should be a corresponding call to biowait() for the same bp.

If B_CALL is set in bp->b_flags, then biodone() calls the callback function specified in bp->b_iodone. The callback function is expected to set the B_DONE flag in bp->b_flags.

If B_CALL is not set in bp->b_flags, then biodone() marks the buffer I/O as completed by setting the B_DONE flag in bp->b_flags. If B_ASYNC is set, then biodone() releases the buf structure and associated buffer pointed to by bp, else it resumes the thread waiting on the corresponding call to biowait()

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock of order >= BUF_HASH_LOCK_ORDER.

WARNINGS

biodone() calls panic() if B_DONE is set in bp->b_flags upon entry.
EXAMPLE

/*
 * As a sanity check, make sure that B_DONE is not set
 * in b_flags before we call biodone(). If B_DONE is
 * set, then we must be holding a stale buf structure.
 */
VASSERT(!(bp->b_flags & B_DONE));

/*
 * Return I/O completion info in the buf structure.
 */
if (transfer_error) {
    bp->b_error  = EIO;
    bp->b_flags |= B_ERROR;
} else {
    bp->b_resid = transfer_residue;
}

/*
 * Complete the buffer I/O transaction. Typically, this
 * results in awaking the thread sleeping in biowait().
 */
biodone(bp);

SEE ALSO

biowait(KER2), buf(KER4)
NAME

biowait(KER2) - Wait for the buffer I/O to complete.

SYNOPSIS

#include <sys/buf.h>

int biowait (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The biowait() kernel function waits for the completion of the buffer I/O specified by bp. A corresponding call to biodone() is required to resume the waiting thread.

RETURN VALUES

biowait() returns the following values:

0 Successful completion.

<>0 Error.

CONSTRAINTS

Must not be called in an interrupt context.
Must not be called while holding a spinlock.

EXAMPLES

int error;
struct buf *bp;
...

/*
 * After starting the I/O request, wait for its completion.
 */
error = biowait(bp);
/
* biowait() returns 0 if the IO completes successfully.
* A non-zero value is returned if an error has been
* encountered, however, the error value returned is not
* always for the IO completion. To get the IO
* completion error that is returned with the buf, we
* need to call geterror().
*/
if (error) {
    error = geterror(bp);
}

SEE ALSO
biodone(KER2), buf(KER4), geterror(KER2)
NAME

brelse(KER2) – Release a buffer to the buffer cache.

SYNOPSIS

#include <sys/buf.h>

void brelse (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The brelse() kernel function releases a buffer to the buffer cache. The buffer header (buf structure) is pointed to by bp and it should have been previously allocated by a call to geteblk(). If there are threads waiting for this or any free buffer in the buffer cache, the waiting threads are awakened by brelse().

The B_BUSY and B_BCACHE flags must be set in bp->b_flags upon entry to brelse().

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock of order >=
BUF_FREE_LOCK_ORDER

SEE ALSO

buf(KER4), geteblk(KER2)
NAME

buf(KER4) – File system buffer header structure for block I/O

SYNOPSIS

#include <sys/buf.h>

DESCRIPTION

The buf structure describes a file system buffer header used for block I/O. The buffer header contains a pointer to the system buffer allocated to the header and specifies control and status information for the I/O transfer to be performed. Block drivers are passed buffer headers through their driver_strategy routines. Buffer headers and their associated system buffers may also be allocated by drivers through geteblk().

The buf structure also describes a buffer header used for raw I/O. The buffer header in this case is formatted by physio() and points to a user buffer, not a system buffer. The user buffer is mapped into kernel space for legacy drivers that specify C_MAP_BUFFER_TO_KERNEL in the d_flags field of their drv_opts_t structure.

Do not depend on the size of the buf structure when writing a driver. In particular, do not allocate a buf structure through kmalloc() and related memory allocation functions. Doing so may invalidate binary compatibility between releases of HP-UX.

STRUCTURE MEMBERS

The buf structure is defined in <sys/buf.h>. The following table lists important fields in the buffer and their types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct buf *</td>
<td>av_back</td>
</tr>
<tr>
<td>struct buf *</td>
<td>av_forw</td>
</tr>
<tr>
<td>char</td>
<td>b_ba</td>
</tr>
</tbody>
</table>
Kernel Reference Pages
Functions and Structures

Table 2-1 Relevant buf Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>b_bcount</td>
</tr>
<tr>
<td>daddr_t</td>
<td>b_blkno</td>
</tr>
<tr>
<td>long</td>
<td>b_bufsize</td>
</tr>
<tr>
<td>dev_t</td>
<td>b_dev</td>
</tr>
<tr>
<td>short</td>
<td>b_error</td>
</tr>
<tr>
<td>long</td>
<td>b_flags</td>
</tr>
<tr>
<td>int *</td>
<td>b_iодone()</td>
</tr>
<tr>
<td>unsigned int</td>
<td>b_resid</td>
</tr>
<tr>
<td>space_t</td>
<td>b_spaddr</td>
</tr>
<tr>
<td>caddr_t</td>
<td>b_un.b_addr</td>
</tr>
</tbody>
</table>

**av_back** **av_forw**

Backward and forward pointers in the buffer headers on the free list where the B_BUSY flag is not sent in b_flags. Drivers can use the av_forw and av_back pointers to maintain a queue of busy buffer headers.

**b_bcount**

The number of bytes to be transferred.

**b_blkno**

Block number of the first logical block to be accessed on the target device. A block contains DEV_BSIZE bytes.

**b_bufsize**

Size of the allocated buffer.

**b_dev**

The dev_t major and minor numbers of the target device.

**b_error**

If the B_ERROR bit is set in b_flags, b_error contains the errno value for the error that occurred. This field is set by the driver before calling biodone() and is obtained by calling geterror().

**b_flags**

Information about the state of the buffer. This value is composed from the following bit flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>If set...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Kernel Reference Pages

#### Functions and Structures

- **B_ASYNC**: Buffer write is synchronous. Do not wait for I/O completion. Mutually exclusive with B_SYNC.

- **B_BCACHE**: The buffer is allocated from the file system buffer cache.

- **B_BUSY**: The buffer is in use.

- **B_CACHE**: The buffer is allocated from the file system buffer cache.

- **B_CALL**: bread() located this buffer in the cache.

- **B_CALL**: iodone() is to call the function pointed to by b_iiodone.

- **B_DELWRI**: Delayed write. Write at exit of avail list processing by the buffer cache management code.

- **B_DONE**: biodone() sets this flag.

- **B_END_OF_DATA**: The buffer transfer has completed.

- **B_ERROR**: An error occurred during the I/O transfer. If the driver sets this flag, it must also set the b_error field with an errno value.

- **B_FSYSIO**: Buffer came from bread() or bwrite().

- **B_INVAL**: The buffer does not contain valid information.

- **B_NDELAY**: Do not retry on failures.

- **B_NOCACHE**: Do not cache data buffer when released.

- **B_PAGEOUT**: This flag is used by the buffer cache management system and should not be touched by a driver.

- **B_PFTIMEOUT**: With this flag set, a driver is expected to return the I/O request with b_error set to EPOWERF if the device...
has experienced a power failure. Drivers typically employ a timeout mechanism to detect a device power failure during an I/O transfer.

- **B_PHYS**
  Indicates the buffer is a user buffer. This flag is normally set by physio().

- **B_PRIVATE**
  Indicates the buffer header is private to a subsystem such as LVM.

- **B_RAW**
  Indicates the buffer header is sent to a character (raw) device.

- **B_READ**
  Data are to be read from the device to host memory. If B_READ is not set, data are to be written from host memory to the device.

- **B_REWRITE**
  This flag is used by the buffer cache management system and should not be touched by a driver.

- **B_SYNC**
  Buffer write is synchronous. Wait for I/O completion. Mutually exclusive with B_ASYNC.

- **B_WANTED**
  One or more threads are sleeping on the buffer header, waiting for the buffer to be freed.

- **B_WRITE**
  A pseudo flag that semantically indicates "not B_READ". The value of B_WRITE is 0; it has no testable bits. To test for a write request, test for the absence of B_READ:

  ```c
  if (!(bp->b_flags & B_READ) )
  ```

  The expression (bp->b_flags & B_WRITE) is always zero.

- **B_WRITEV**
  This flag is used by LVM when attempting to correct disk soft errors. and should not be touched by a driver.

- **b_iiodone**
  Pointer to a function that iodone() calls to complete
the I/O request if the B_CALL flag is set in b_flags. The
function takes a pointer to the buffer header as its
argument and is expected to set the B_DONE flag in
b_flags.

b_resid  Number of bytes remaining to be transferred; usually
set to zero after a successful transfer. The
driver_strategy routine sets this field before calling
biodone().

b_spaddr  Space ID of the buffer specified by the buffer header.
Do not assume this value to be KERNELSPACE.

b_un.b_addr  Virtual address of the buffer specified by the buffer
header. The buffer may be mapped in kernel space or it
may be mapped in user space. If mapped in user space,
b_spaddr must be used with b_addr to form the global
virtual address in order to access the buffer - see
privlbcopy().

SEE ALSO

biodone(KER2), biowait(KER2), brelse(KER2), geteblk(KER2),
geterror(KER2), physio(KER2), privlbcopy(KER2)
**NAME**

`busywait(KER2)` - Wait at least $t$ microseconds by spinning processor cycles.

**SYNOPSIS**

```c
#include <sys/kern_svc.s.h>
#include <sysio.h>

void busywait (ulong_t t);
```

**PARAMETERS**

$t$ The wait time in microseconds.

**DESCRIPTION**

The `busywait()` kernel function waits at least $t$ microseconds by spinning processor cycles. The processor does no useful work while busy waiting, so $t$ should be kept as short as possible.

`busywait()` does not block (i.e., sleep) and can be called in an interrupt context or while holding a spinlock. If `busywait()` is entered with external interrupts enabled, it is possible for an interrupt to occur and make the actual wait time far exceed the specified $t$ microseconds.

**RETURN VALUES**

None.

**CONSTRAINTS**

**EXAMPLES**

**SEE ALSO**
NAME

bzero(KER2) - Fill a kernel buffer with zeros.

SYNOPSIS

#include <sys/kern_svcs.h>

int bzero (caddr_t addr, size_t n);

PARAMETERS

addr Address of kernel buffer.

n Number of bytes to be zeroed.

DESCRIPTION

The bzero() kernel function writes n contiguous bytes of zero, starting at the kernel address specified by addr.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

bcopy (KER2)
NAME

`copyin(KER2)` - Copy data from a user buffer to a kernel buffer

SYNOPSIS

```c
#include <sys/kern_svc.h>

int copyin (caddr_t from_user, caddr_t to_kernel, size_t n);
```

PARAMETERS

- `from_user` : Source user space address.
- `to_kernel` : Destination kernel space address.
- `n` : Number of bytes to copy.

DESCRIPTION

The `copyin()` kernel function copies `n` bytes of data from the user space address `from_user` to the kernel space address `to_kernel`. The call to `copyin()` must be made while executing in the user context; that is, while executing in the top half of the driver where the user invokes the driver via a system call such as `ioctl()`.

`copyin()` may block (i.e., sleep), so it must not be called while holding a spinlock.

RETURN VALUES

`copyin()` returns the following values:

- `0` : Successful completion.
- `<>0` : Error.

CONSTRAINTS

- Must be called in an interrupt context.
- Must be called while holding a spinlock.
WARNINGS

The kernel stack is limited in size. If the buffer to be copied is larger than 128 bytes, avoid allocating buffer space on the kernel stack; instead, allocate the buffer from kernel memory (e.g., by calling `kmalloc()`). When large buffers are allocated on the kernel stack, the kernel stack may overflow and cause the kernel to panic.

EXAMPLE

```c
char my_buff[128];
/*
 * Copy from the user buffer to my_buff[] on the kernel stack.
 * Note that buffers larger than 128 bytes should be
 * allocated from kernel memory by calling kmalloc().
 */
if (copyin(user_buf, my_buf, sizeof(my_buf))) {
    return EFAULT;
}
```

SEE ALSO

`bcopy(KER2), copyout(KER2), privlbcopy(KER2), uiomove(KER2)`
NAME

`copyout(KER2)` - Copy data from a kernel buffer to a user buffer

SYNOPSIS

```c
#include <sys/kern_svcs.h>

int copyout (caddr_t from_kernel, caddr_t to_user, size_t n);
```

PARAMETERS

- `from_kernel`  Source kernel space address.
- `to_user`  Destination user space address.
- `n`  Number of bytes to copy.

DESCRIPTION

The `copyout()` kernel function copies `n` bytes of data from the kernel space address `from_kernel` to the user space address `to_user`. The call to `copyout()` must be made while executing in the user context; that is, while executing in the top half of the driver where the user invokes the driver via a system call such as `ioctl()`.

`copyout()` may block (i.e., sleep), so it must not be called while holding a spinlock.

RETURN VALUES

`copyout()` returns the following values:

- `0`  Successful completion.
- `<>0`  Error.

CONSTRAINTS

Must be called in an interrupt context.

Must not be called while holding a spinlock.
WARNINGS

The kernel stack is limited in size. If the buffer to be copied is larger than 128 bytes, avoid allocating buffer space on the kernel stack; instead, allocate the buffer from kernel memory (e.g., by calling kmalloc()). When large buffers are allocated on the kernel stack, the kernel stack may overflow and cause the kernel to panic.

EXAMPLE

```c
char my_buff[128];

/*
 * Copy to the user buffer from my_buff[] on the kernel stack.
 * Note that buffers larger than 128 bytes should be
 * allocated from kernel memory by calling kmalloc().
 */
if (copyout(my_buf, user_buf, sizeof(my_buf))) {
    return EFAULT;
}
```

SEE ALSO

bcopy(KER2), copyin(KER2), privlbcopy(KER2), uiomove(KER2)
**NAME**

cspinlock(KER2) – Conditionally acquire (lock) a spinlock.

**SYNOPSIS**

```c
#include <sys/spinlock.h>

int cspinlock (lock_t * lock);
```

**PARAMETERS**

- `lock` Pointer to a `lock_t` structure.

**DESCRIPTION**

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The `cspinlock()` kernel function attempts to conditionally acquire (lock) a spinlock pointed to by `lock`. The calling thread does not spin if the spinlock is currently owned.

**RETURN VALUES**

`cspinlock()` returns the following values:

- 0 The lock is already owned by this processor or another processor.
- 1 The lock has been acquired by this processor.

**CONSTRAINTS**
while (list_entry != list_head) {
    /*
    * Get the next list entry protected by mydrv_lock_2.
    */
    spinlock(mydrv_lock_2);
    list_entry = list_head->next_entry;
    if (list_entry == list_head) {
        spinunlock(mydrv_lock_2);
        break; /* at end of list */
    }

    /*
    * We need to acquire another spinlock, but we are
    * acquiring the two locks in reverse order. To avoid
    * a deadlock, we conditionally attempt to acquire
    * the next spinlock with cspinlock(). If already
    * owned, we must release the other spinlock.
    *
    * Note: cspinlock() does not check the lock order.
    */
    if (!cspinlock(mydrv_lock_1)) {
        /*
        * Failed to acquire the next spinlock.
        * As a sanity check, assert that we are not
        * on the processor that owns the spinlock.
        */
        VASSERT(!owns_spinlock(mydrv_lock_1));
        /*
        * Release mydrv_lock_2 to allow the processor
        * holding mydrv_lock_1 to make forward progress.
        */
        spinunlock(mydrv_lock_2);
        /*
        * Get the two locks in the correct lock order.
        */
        spinlock(mydrv_lock_1);
        spinlock(mydrv_lock_2);
    }

    /*
    * We now have both locks. Check to be sure the
    * list_head did not change if we had to release
    * mydrv_lock_2 to avoid a deadlock.
    */
if (list_entry != list_head->list_entry) {
    /*
     * Release the two locks and try again.
     */
    spinunlock(mydrv_lock_2);
    spinunlock(mydrv_lock_1);
    continue;
}

/*
 * Process the list entry.
 */
...

/*
 * Release the two locks and get the next entry.
 */
spinunlock(mydrv_lock_2);
spinunlock(mydrv_lock_1);
} /* end of while */

SEE ALSO

alloc_spinlock(KER2), dealloc_spinlock(KER2),
owns_spinlock(KER2), spinlock(KER2), spinunlock(KER2)
NAME

dealloc_spinlock(KER2) - Deallocate a spinlock resource.

SYNOPSIS

#include <sys/spinlock.h>

void dealloc_spinlock (lock_t * lock);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitives used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The dealloc_spinlock() kernel function deallocates a spinlock resource, pointed to by lock, which was previously allocated by alloc_spinlock().

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

alloc_spinlock(), cspinlock(), get_sleep_lock(), owns_spinlock(), spinlock(), spinunlock()
NAME
FREE(KER2) Kernel Macro – Kernel macro to deallocate kernel memory.

SYNOPSIS
#include <sys/malloc.h>
define FREE (addr, type)

PARAMETERS
addr Address of the kernel memory previously allocated by MALLOC().
type The memory allocation pool type.

DESCRIPTION
The MALLOC() and FREE() kernel macros are deprecated interfaces and may be obsoleted in a future release of HP-UX. Use the kernel functions kmalloc() and kfree() in place of MALLOC() and FREE().

The FREE() kernel macro deallocates (frees) kernel memory previously allocated by MALLOC(). The addr parameter must be the address returned by MALLOC() and type must be the same type passed to MALLOC().

CONSTRAINTS

EXAMPLES
struct buf * bp;

/*
 * Allocate a buf structure for internal use.
 */
MALLOC(bp, struct buf *, sizeof(struct buf), M_IOSYS, M_NOWAIT);
...

56 Chapter 2
/*  
*  When done with the buf structure, release it.  
*/  
FREE(bp, M_IOSYS);

SEE ALSO

kmalloc(KER2), kfree(KER2), MALLOC(KER2)
NAME

get_sleep_lock(KER2) – Acquire a sleep queue spinlock.

SYNOPSIS

#include <sys/spinlock.h>

lock_t * get_sleep_lock (caddr_t chan);

PARAMETERS

chan Channel parameter passed to sleep() and wakeup().

DESCRIPTION

The get_sleep_lock() kernel function acquires the spinlock that protects the sleep queue associated with the channel chan. Different values of chan may map onto different sleep queues. A thread calls get_sleep_lock() before it calls sleep(). The sleep queue spinlock associated with chan is released by the corresponding call to sleep().

get_sleep_lock() synchronizes the race condition between sleep() and wakeup(). It is possible for wakeup() to be called on another processor before a thread can be put to sleep. However, the corresponding wakeup() must acquire the sleep queue spinlock held by the thread being put to sleep, thus it spins and waits. When wakeup() does acquire the spinlock, the thread will be on the sleep queue and ready to be awakened.

get_sleep_lock() may optionally be used as a synchronization wrapper around wakeup(). For example, consider the following code:

    (void) get_sleep_lock (wait_chan);
    start_async_activity();
    activity_count++;
    (void) sleep (wait_chan, PRIBIO);

Notice that activity_count is protected by a sleep queue spinlock which is released by sleep(). When the async activity completes, the corresponding completion routine calls get_sleep_lock() before it decrements activity_count and calls wakeup()

    sleep_lock = get_sleep_lock (wait_chan);
    if (activity_count) {


activity_count--;  
(void) wakeup(wait_chan);
}  
spinunlock(sleep_lock);

wakeup() can handle the case where the sleep queue spinlock is held by the caller upon entry. The sleep queue spinlock must be explicitly released by a call to spinunlock() when used in the above manner.

RETURN VALUES

get_sleep_lock() returns a pointer to the lock_t structure which contains the spinlock that protects the sleep queue.

CONSTRAINTS

Must not be called while holding a spinlock of order >= SLEEP_Q_LOCK_ORDER.

EXAMPLES

/*
 * The sleeping thread waits on a channel by first calling
 * get_sleep_lock(), then it starts an asynchronous activity
 * and finally calls sleep() which releases the sleep lock.
 */
(void) get_sleep_lock(wait_chan);
start_async_activity();
(void) sleep(wait_chan, PRIBIO);
...

/*
 * When the asynchronous activity completes (typically in an
 * interrupt service routine), the sleeping thread is awakened.
 */
(void) wakeup(wait_chan);

SEE ALSO

sleep(KER2), wakeup(KER2)
NAME

getc(KER2) – Manipulate characters on a clist

SYNOPSIS

#include <sys/clist.h>

int getc (struct clist * list);

PARAMETERS

list Pointer to a clist.

DESCRIPTION

The getc() kernel function returns the next character in the clist list.

RETURN VALUES

getc() returns the following values:

c The next character in the clist.

-1 If no characters are in the clist.

CONSTRAINTS

EXAMPLES

void
catq(struct clist * from,
    struct clist * to)
{
    int c;

    while ((c = getc(from)) >= 0)
    (void) putc(c, to);
SEE ALSO

putc(KER2), getcb(KER2), getcf(KER2), putcb(KER2), putcf(KER2)
NAME

getcb (KER2) – Manipulate cblocks on a dist

SYNOPSIS

#include <sys/clist.h>

struct cblock * getcb (struct clist * list);

PARAMETERS

list Pointer to a clist.

DESCRIPTION

getcb() removes the last cblock from the clist pointed to by list.

RETURN VALUES

getcb() returns the following values:

<>NULL Successful completion. The value is a pointer to the
cblock that was removed.

NULL Error.

CONSTRAINTS

EXAMPLE

int
mydev_open(dev_t dev, int flag)
{
    struct cblock *cp;
    struct mydevice dv;

    /* Flush out the device buffer */
    while ((cp = getcb(&dv->queue)) != NULL)
        putcf(cp);

    ...

    return 0; // Assume return value for mydev_open
}
SEE ALSO

putc(KER2), getc(KER2), getcf(KER2), putc(KER2), putcf(KER2)
NAME
getcf(KER2) - Get a cblock on the cblock free list

SYNOPSIS
#include <sys/clist.h>
struct cblock * getcf (void);

PARAMETERS
None

DESCRIPTION
getcf() gets the next free cblock on the cfreelist.

RETURN VALUES
getcf() returns the following values:
<>NULL Successful completion. The value is a pointer to the
cblock.
NULL Error.

CONSTRAINTS

SEE ALSO
getc(KER2), getcb(KER2)
NAME

geteblk(KER2) - Allocate a buffer from the buffer cache.

SYNOPSIS

#include <sys/buf.h>

struct buf * geteblk (int size);

PARAMETERS

size The size in bytes of the requested block.

DESCRIPTION

The geteblk() kernel function allocates a buffer large enough to contain size bytes of data from the buffer cache and returns a pointer to the buffer header. The size parameter must be <= MAXBSIZE defined in <sys/param.h>.

Fields in the buf structure returned are initialized as follows:

- b_flags is set with B_BUSY, B_BCACHE and B_INVAL flags
- b_un.b_addr contains the virtual address of the allocated buffer
- b_spaddr contains the space address (space ID) of the allocated buffer

Drivers do not normally allocate buffers. However, to implement special features, such as ioctl() commands, the driver may need its own buffer space. Buffers allocated by calling geteblk() should be used for short durations and released by calling brelse() when no longer needed.

RETURN VALUES

geteblk() returns a pointer to the allocated buffer header.

CONSTRAINTS

Must not be called in an interrupt context.
 Must not be called while holding a spinlock.
EXAMPLE

struct buf * bp;

/*
 * Allocate a buffer that can store a page of data.
 */
bp = geteblk(PAGESIZE);
VASSERT(bp);

SEE ALSO

brelse(KER2), buf(KER2)
NAME

gerror(KER2) – Return the error number from the buffer header.

SYNOPSIS

#include <sys/buf.h>

int gerror (struct buf *bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

gerror() returns the error number from the b_error field of the buffer header structure pointed to by bp. If B_ERROR is not set in bp->b_flags, the I/O is assumed to have completed successfully.

RETURN VALUES

gerror() returns the following values:

0 I/O completed successfully.
<>0 Error number.

CONSTRAINTS

EXAMPLE

int error;
struct buf *bp;
...

/*
 * After starting the I/O request, wait for its completion.
 */
error = biowait(bp);
/*
 * biowait() returns 0 if the IO completes successfully.
 * A non-zero value is returned if an error has been
 * encountered, however, the error value returned is not
 * always for the IO completion. To get the IO
 * completion error that is returned with the buf, we
 * need to call geterror().
 */
if (error) {
    error = geterror(bp);
}

SEE ALSO

biowait(KER2), buf(KER4)
NAME

gsignal(KER2) - Send the specified signal to all processes in a process group

SYNOPSIS

#include <h/proc_iface.h>

void gsignal (pid_t pgrp, int sig);

PARAMETERS

pgrp Process group identifier.
sig Signal number.

DESCRIPTION

The gsignal() kernel function sends the signal sig to all processes that have pgrp as their process group identifier.

The pgrp parameter is returned by the kernel function p_grp().

Signal numbers are defined in <sys/signal.h>

RETURN VALUES

None.

CONSTRAINTS

EXAMPLES

#include <sys/user.h>
#include <sys/signal.h>
#include <h/proc_iface.h>

/
* Signal all processes in the current process group.
*/
gsignal(p_pgrp(u.u_procp), SIGIO);
SEE ALSO

p_pgrp(KER2), psignal(KER2)
NAME

`iovec(KER4)` - Data buffer descriptor for character I/O and WSIO CDIO mapping services.

SYNOPSIS

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

DESCRIPTION

The `iovec` kernel structure points to the data buffer for character I/O. It can be declared and used to map a kernel data buffer for an I/O bus master and it is a member of the `uio` structure used by `uiomove()`.

STRUCTURE MEMBERS

The `iovec` structure is defined in `<sys/uio.h>`. It has the following fields. Their data types are shown in the table below. Drivers must not modify any fields of this structure when using `uiomove()`. When this structure is used for mapping with a WSIO CDIO mapping service, the driver initially sets up `iov_base` and `iov_len`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>caddr_t</code></td>
<td><code>iov_base</code></td>
</tr>
<tr>
<td><code>size_t</code></td>
<td><code>iov_len</code></td>
</tr>
</tbody>
</table>

- `iov_base` - The address of the user's buffer for `uiomove` or kernel buffer for WSIO CDIO mapping services.
- `iov_len` - The number of bytes to be transferred.

SEE ALSO

`uio(KER4)`, `wsio_fastmap(WSIO3)`, `wsio_map(WSIO3)`, `wsio_remap(WSIO3)`
NAME

isc_table_type(KER4) – ISC table entry structure

SYNOPSIS

#include <sys/io.h>

DESCRIPTION

Each interface card has an associated Interface Select Code (ISC) entry. Each ISC entry, defined as an isc_table_type kernel structure, is used by WSIO to maintain interface driver information.

An interface driver gets information specific to each of its interface cards by referencing the appropriate ISC entry.

A device driver uses wsio_get_isc() service to obtain a pointer to the ISC entry for its corresponding interface driver. The device driver passes the isc pointer to the interface driver through the ifsw table pointed to by the isc.

The I/O Switch Tables

The I/O system supports two I/O interface switch tables through fields in the ISC structure. isc->gfsw is intended for use by the system. isc->ifsw is available for communication between drivers and interface drivers. The fields in both are filled in by each interface driver, usually during the boot process, in accordance with its own needs. The field values default to NULL.

Generic Function Switch

The generic function switch, isc->gfsw, defined in <sys/io.h>, is intended for system-to-interface driver communication, not device driver-to-interface driver communication. The table consists of pointers to two function routines:

- isc->gfsw.init points to a driver-defined interface initialization routine that is called after all calls to driver_attach() functions have been made.
- isc->gfsw.diag points to a driver-defined interface diagnostic routine whose usage is currently not implemented.

Interface Function Switch

The interface function switch, isc->ifsw, is intended for device driver-to-interface driver communication. It is through this table that a device driver calls its corresponding interface driver. The table is defined
and initialized by the interface driver.

For an example of an interface switch table, see the `drv_table_type` structure in `<sys/hpibio.h>`.

**STRUCTURE MEMBERS**

The `isc_table_type` structure is defined in `<sys/io.h>`. The following is a list of the fields that drivers normally use. Their data types are shown in the tables below. Some are read-only; others are set and used by the driver.

<table>
<thead>
<tr>
<th>Table 2-3 Driver Relevant <code>isc_table_type</code> Structure Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td><code>struct gfsw *</code></td>
</tr>
<tr>
<td><code>caddr_t</code></td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td><code>caddr_t</code></td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td><code>caddr_t</code></td>
</tr>
<tr>
<td><code>struct buf *</code></td>
</tr>
</tbody>
</table>

- `bus_type`: Type of I/O bus for the interface card. For example, `PCI_BUS` for PCI interface cards.
- `ftn_no`: For multifunction EISA cards, contains the function number associated with this ISC.
- `gfsw`: Pointer to the generic interface driver function switch table.
- `if_drv_data`: For driver specific information.
- `if_id`: Contains this interface card’s hardware ID.
if_info Pointer to a wsio_if_info structure defined in <sys/wsio.h> (also defined identically as an eisa_if_info structure in <sys/eisa.h>)

if_info->flags Flags indicating the result of an interface driver attempting to claim an interface card. Currently defined values are:

HAS_IOCHKERR The card has an I/O check error.
INITIALIZED An interface driver’s attach routine has successfully initialized the card. This flag is set by the isc_claim() function.
INIT_ERROR An interface driver’s attach routine tried to claim this card, but failed.
IS_ISA_CARD This card is an ISA card.
NO_ALLOC_GDD Reserved.
SLOT_ADDR_VALID Reserved.

if_reg_ptr Pointer to the base of the interface card’s memory mapped registers. After initial configuration, i.e., when your interface driver’s attach routine is called, if your driver claims the card you may use this field as desired. For example, if your PCI driver memory space is not mapped due to size constraints, you can call map_mem_to_host() and store the returned virtual address from that call in this field.

Table 2-4 Other Generally Available isc_table_type Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>caddr_t</td>
<td>buffer</td>
</tr>
<tr>
<td>int *</td>
<td>card_ptr</td>
</tr>
<tr>
<td>int</td>
<td>count</td>
</tr>
<tr>
<td>short</td>
<td>dma_reserved</td>
</tr>
<tr>
<td>short</td>
<td>dma_active</td>
</tr>
<tr>
<td>struct dma_parms *</td>
<td>dma_parms</td>
</tr>
</tbody>
</table>

74 Chapter 2
Table 2-4 Other Generally Available \texttt{isc\_table\_type} Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{caddr_t}</td>
<td>\texttt{ifsw}</td>
</tr>
<tr>
<td>\texttt{struct sw_intloc}</td>
<td>\texttt{intloc1}</td>
</tr>
<tr>
<td>\texttt{char}</td>
<td>\texttt{int_lvl}</td>
</tr>
<tr>
<td>\texttt{char}</td>
<td>\texttt{my_address}</td>
</tr>
<tr>
<td>\texttt{int}</td>
<td>\texttt{resid}</td>
</tr>
<tr>
<td>\texttt{int (*)()}</td>
<td>\texttt{transaction_proc}</td>
</tr>
<tr>
<td>\texttt{ENUM TFR_type}</td>
<td>\texttt{transfer}</td>
</tr>
</tbody>
</table>

\textbf{owner} Pointer to the \texttt{buf} structure being used by this interface for transfer. Interface drivers use this field to control transfers in a \texttt{driver\_isr()} because an \texttt{isc\_table\_type} pointer is usually the first parameter passed into the routine which was registered with "isrlink()".

\textbf{buffer} Can be used as desired.

\textbf{card\_ptr} Pointer to a specific area of the interface card's register space. This field can be used by the interface driver to point to additional registers. It is set by the interface driver.

\textbf{count} Can be used as desired.

\textbf{dma\_reserved} Can be used as desired.

\textbf{dma\_active} Can be used as desired.

\textbf{dma\_parms} Can be used as desired. Typically used by the \texttt{driver\_isr()} routine to obtain information from the \texttt{dma\_parms} structure used in the current transfer.

\textbf{ifsw} Interface driver entry-point switch. It is set by the interface driver and is intended to be an operational interface between a device driver and its interface driver.

\textbf{intloc1} Can be used as desired. Used for software triggers.
Kernel Reference Pages
Functions and Structures

int_lvl Can be used as desired. Typically contains the interface card’s interrupt level.

my_address Can be used as desired. Usually contains the interface card’s bus address.

resid Can be used as desired. DMA handlers can use to hold the residual count from a data transfer.

transaction_proc Can be used as desired.

transfer Can be used as desired.

### Table 2-5

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct isc_table_type*</td>
<td>next_ftn</td>
</tr>
<tr>
<td>void *</td>
<td>drv_hdr</td>
</tr>
<tr>
<td>void *</td>
<td>card_node</td>
</tr>
<tr>
<td>io_map_cntl_t *</td>
<td>map_funcs</td>
</tr>
<tr>
<td>struct map_mem *</td>
<td>mem_map</td>
</tr>
<tr>
<td>struct dma_funcs *</td>
<td>dma_funcs</td>
</tr>
<tr>
<td>struct eisa_iomap</td>
<td>iomap_funcs</td>
</tr>
<tr>
<td>int</td>
<td>if_inited</td>
</tr>
<tr>
<td>intptr_t</td>
<td>eim_control</td>
</tr>
<tr>
<td>intptr_t</td>
<td>eim</td>
</tr>
</tbody>
</table>

**WARNINGS**

The structure field usage discussed in this man page may change with subsequent releases of HP-UX.
NAME

kernel_iomap(KER2) – Map physical space to host virtual space with user read/write protection.

SYNOPSIS

#include <sys/iomap.h>

caddr_t kernel_iomap (caddr_t virt_addr, caddr_t phys_addr, pgcnt_t count, prot_t prot);

PARAMETERS

virt_addr NULL or equivalent to physical address
phys_addr The start of the host physical address (not the bus relative address) for an area of memory on an I/O bus (EISA, for example).
count The length of the mapping in pages (4 Kbyte pages).
prot The protection to be used for the mapping which should be PROT_URW.

DESCRIPTION

The kernel routine kernel_iomap() is used to map physical space onto a host virtual address for cards with memory components that need to be mapped in for user read/write access. It maps the space with user read/write protection.

RETURN VALUES

The kernel_iomap() routine returns the following values:

<>NULL Successful completion. The value is the host virtual address for accessing the space specified by the parameters.
NULL Error.
CONSTRAINTS

EXAMPLES

/*
** example maps 1 page of I/O space at 0xf0000000 with
** user read/write protections (PROT_URW)
*/

#include "/usr/conf/h/mman.h"
#include "/usr/conf/machine/psl.h"

{
    int sm;
    char byte;
    caddr_t mapped_addr;
    caddr_t phys_addr = 0xf0000000;
    u_int count = 1;

    /* map the space to the driver */
    mapped_addr =
        kernel_iomap(NULL,phys_addr,count,PROT_URW);
    if (mapped_addr == NULL)
        goto map_failed; /* bailout code for error */
    /* grant kernel access to user mapped space */
    sm=rsm(PSW_P);
    byte = *mapped_addr; /* read first byte of space */
    /* other access to user mapped I/O space */
    /* deny kernel access to space */
    if (sm & PSW_P)
        ssm(PSW_P);
    /*
    ** Note - even though the space is mapped with user
    ** read/write protections, this does NOT mean that a
    ** user process can access the space. Your driver
    ** should call user_iomap, to map the space to a
    ** given process.
    */
}

Chapter 2
WARNINGS

The use of `kernel_iomap()` which uses PROT_URW, effectively disables kernel mode access of the mapped space. If the driver requires access to the mapped space then the `rsm(PSW_P)` and `ssm(PSW_P)` instructions must be used because the I/O pages are mapped with a non-zero protection ID.

In the case of EISA (not ISA) cards, the EISA IO space is mapped (prior to calling the driver attach routine) with PROT_KRW. This mapping is for one page (4 Kbytes). If you desire user access to EISA IO space you will need to first unmap `isc->if_reg_ptr` with a call to `unmap_mem_from_host()`, and then remap the space with `kernel_iomap_public()`. A similar situation exists with the PCI bus memory space.

Hardware disables caching for all EISA data accesses, including EISA memory space.

`kernel_iomap()` will return NULL if a request overlaps an existing mapping. If the request exactly overlays (same start address and size) an existing mapping, the mapping services will return the host virtual address; n.b., second and subsequent mappings inherit the protection values of the first mapping call.

SEE ALSO

`kernel_iomap_public(KER2)`, `kernel_iounmap(KER2)`, `user_iomap(KER2)`, `user_iounmap(KER2)`
NAME

kernel_iomap_public(KER2) – Map physical bus to host virtual space with no protection.

SYNOPSIS

#include <sys/iomap.h>

#pragma kernel_iomap_public (caddr_t virt_addr, caddr_t phys_addr, pgcnt count, prot_t prot);

PARAMETERS

virt_addr NULL or equivalent to physical address
phys_addr The start of the host physical address (not the bus relative address) for an area of memory on an I/O bus (EISA, for example).
count The size of the memory space in pages (4 Kbyte pages).
prot The protection to be used for the mapping. Should be PROT_URW.

DESCRIPTION

The kernel routine kernel_iomap_public() is used to map physical space onto a host virtual address for cards with memory components that need to be mapped in for user read/write access. When called with prot = PROT_URW, it maps the space with no protection bits set. The behavior of this call is very similar to kernel_iomap() with the exception that drivers can access the mapped pages without using the rsm(PSW_P) and ssm(PSW_P) instructions.

RETURN VALUES

kernel_iomap_public() routine returns the following values:

<>NULL Successful completion. The value is the host virtual address for accessing the space specified by the parameters.
EXAMPLES

/*
 ** example maps 1 page of I/O space at
 ** 0xf0000000 with no protection
 */

#include <sys/mman.h>

{
    int sm;
    char byte;
    caddr_t mapped_addr;
    caddr_t phys_addr = 0xf0000000;
    u_int count = 1;

    /* map the space to the driver */
    mapped_addr =
        kernel_iomap_public(NULL,phys_addr,count,PROT_URW);
    if (mapped_addr == NULL)
        goto map_failed; /* bailout code for error */
    byte = *mapped_addr; /* read first byte of space */
    /* other access to user mapped I/O space */
    /*
    ** Note - even though the space is mapped with user
    ** read/write protections, this does NOT mean that a
    ** user process can access the space. Your driver
    ** should call user_iomap, to map the space to a
    ** given process. Alternatively, the user process
    ** can call the iomap driver
    .
    */

}
WARNINGS

When called with PROT_URW protection mode (prot parameter), this call creates an unprotected mapping which allows I/O access from unauthorized processes.

This call should be used with extreme caution as an unauthorized process could read/write to any I/O space register mapped in this manner and crash/hang the system.

In the case of EISA (not ISA) cards, the EISA IO space is mapped (prior to calling the driver attach routine) with PROT_KRW. This mapping is for one page (4 Kbytes). If you desire user access to EISA IO space you will need to first unmap isc->if_reg_ptr with a call to unmap_mem_from_host(), and then remap the space with kernel_iomap_public(). A similar situation exists for PCI memory space.

Hardware disables caching for all EISA data accesses, including EISA memory space.

kernel_iomap_public() will return NULL if a request overlaps an existing mapping. If the request exactly overlays (same start address and size) an existing mapping, the mapping services will return the host virtual address requested.

SEE ALSO

kernel_iounmap(KER2), unmap_mem_from_host(KER2), user_iomap(KER2), user_iounmap(KER2)
NAME

`kernel_iounmap` (KER2) - Unmap physical bus from host virtual space

SYNOPSIS

```c
#include <sys/iomap.h>

int kernel_iounmap (caddr_t virt_addr, pgcnt_t count);
```

PARAMETERS

- `virt_addr` - The virtual address previously obtained with a `kernel_iomap()` or `kernel_iomap_public()` call.
- `count` - The size of the memory space in pages (4 KByte pages).

DESCRIPTION

The kernel routine `kernel_iounmap()` is used to unmap previously mapped virtual space.

RETURN VALUES

The `kernel_iounmap()` routine returns the following values:

- `1` - Error.
- `0` - Success.

CONSTRAINTS

SEE ALSO

- `kernel_iomap(KER2)`, `kernel_iomap_public(KER2)`,
- `user_iomap(KER2)`, `user_iounmap(KER2)`
NAME
kfree(KER2) – Deallocate kernel memory

SYNOPSIS

#include <sys/malloc.h>

void kfree (char * addr, int type);

PARAMETERS

addr Address of the kernel memory previously allocated by a call to kmalloc().

type The memory allocation pool type.

DESCRIPTION

The kfree() kernel function deallocates (frees) kernel memory previously allocated by a call to kmalloc(). The addr parameter must be the address returned by kmalloc() and type must be the same type passed to kmalloc().

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

kmalloc(KER2)
NAME

kmalloc(KER2) - Allocate kernel memory

SYNOPSIS

#include <sys/malloc.h>

caddr_t kmalloc (unsigned long size, int type, int flags);

PARAMETERS

size    The number of bytes (size) of kernel memory to allocate.
type    The memory allocation pool type.
flags   Flag to indicate the caller cannot block and wait for kernel memory availability.

DESCRIPTION

The kmalloc() kernel function allocates size bytes of kernel memory from the pool type specified. The memory allocated will always be contiguous in the virtual address space, but may be discontiguous in the physical address space. If size is greater than PAGESIZE, the physical pages allocated will probably not be physically contiguous.

Drivers should allocate memory from one of the following pool types: M_IOSYS or M_DMA. The M_IOSYS pool is for general I/O purposes; the M_DMA pool is for DMA purposes.

The flags parameter may optionally have the M_NOWAIT flag set. If M_NOWAIT is set and no memory is available from the requested pool type, the functions will return NULL. Without M_NOWAIT set, the caller can be blocked and made to wait for memory to become available.

The M_NOWAIT flag must be set if kmalloc() is:

- called in an interrupt context, or
- called while holding a spinlock.

If M_NOWAIT is set, the caller must be prepared to handle the case where no kernel memory has been allocated.
M_WAITOK should be passed as the flags parameter if the caller is able to block and wait for memory to become available.

RETURN VALUES

kmalloc() returns the following values:

<>NULL Virtual address of kernel memory allocated.
NULL No kernel memory has been allocated. NULL is returned only when the M_NOWAIT flag is set, otherwise the caller is blocked and waits until memory becomes available.

CONSTRAINTS

If the M_NOWAIT flag is not set:

• the caller must not be in an interrupt context, and
• no spinlocks can be held

SEE ALSO

kfree(KER2)
NAME

Ktimeout(KER2) – Execute a callout function after a specified length of time at driver level interrupt priority.

SYNOPSIS

#include <sys/callout.h>
#include <sys/param.h>

struct callout * Ktimeout (int (*func)(), caddr_t arg, int t,
                         void * dummy);

PARAMETERS

func                   Function to execute when the time value t expires.
arg                    Argument passed to the callout function func.
t                     Time value in number of clock ticks.
dummy                  Must be NULL.

DESCRIPTION

The Ktimeout() kernel function executes the specified callout function func after t clock ticks have expired. Execution of func takes place in an interrupt context at priority level 5 (driver level interrupt priority) where external interrupts to the processor are disabled. Drivers are encouraged to use the preferred interface timeout() where func is scheduled to execute at priority level 2.

The func parameter is a pointer to a function that takes one argument. Although the prototype declares the function to return an int value, the kernel does not make use of the return value.

The arg parameter is passed as the one argument to func.

The t parameter specifies the number of clock ticks to wait before calling func. To express time in seconds, multiply t by HZ, where HZ is defined as the number of clock ticks per second in <sys/param.h>.

The call to Ktimeout() returns immediately without waiting for the time value t to expire. The timeout can be cancelled by making a
corresponding call to `untimeout()`.

**RETURN VALUES**

`Ktimeout()` returns a pointer to a callout structure.

**CONSTRAINTS**

Must not be called while holding a spinlock of order >= `CALLOUT_LOCK_ORDER`.

**WARNINGS**

Callout resources are not dynamically expandable. Each call to `Ktimeout()` allocates a callout resource, and the resource is not released until the time value expires or the timeout is cancelled. The kernel may panic if no callout resources are available.

**EXAMPLES**

```c
/*
 * Set a timeout to call my_timeout_func() passing my_arg
 * after 5 seconds have expired.
 */
(void)Ktimeout(my_timeout_func, my_arg, 5*HZ, NULL);
...

static int
my_timeout_func(caddr_t arg)
{
    ...
}
```

**SEE ALSO**

`timeout(KER2)`, `untimeout(KER2)`
NAME

`ldsid(KER2)` - Return (load) the space ID (SID) for a kernel virtual address.

SYNOPSIS

```c
#include <sys/kern_svc.h>

space_t ldsid (caddr_t addr);
```

PARAMETERS

`addr`  
Kernel virtual address

DESCRIPTION

The `ldsid()` kernel function returns the space ID (SID) for the kernel virtual address `addr`. The SID and virtual address are combined to form the global virtual address.

`ldsid()` may only be used with kernel virtual addresses. Do not call `ldsid()` for a user space address.

RETURN VALUES

`ldsid()` returns the space ID (SID) for a kernel virtual address.

CONSTRAINTS

EXAMPLES

```c
/*
 * Copy data into the buffer specified through a buf (bp).
 * The b_spaddr and b_un.b_addr fields may contain the SID
 * and virtual address of a user space buffer. The from_addr
 * is a kernel virtual address and ldsid() returns its SID.
 */
privlbcopy(ldsid(from_addr), from_addr,
            bp->b_spaddr, bp->b_un.b_addr,
            sizeof(*from_addr));
```
SEE ALSO

privlbcopy(KER2)
NAME

major(KER2) - Extract the major number from a device number

SYNOPSIS

#include <sys/sysmacros.h>

#define major(x) ((int)(((unsigned)(x)>>24)&0xff))

PARAMETERS

x A dev_t device number.

DESCRIPTION

major() is a macro that returns the major number from the device number.

CONSTRAINTS

SEE ALSO

minor(KER2)
NAME

makedev(KER2) - Make a device number from major and minor numbers

SYNOPSIS

#include <sys/sysmacros.h>

#define makedev(x, y) ((dev_t)((x)<<24) | (y & 0xffffffff))

PARAMETERS

x  A major number.
y  A minor number.

DESCRIPTION

The makedev() kernel macro builds a device number from major and minor numbers.

CONSTRAINTS

SEE ALSO

major(KER2), minor(KER2)
NAME

MALLOC(KER2) – Kernel macro to allocate kernel memory

SYNOPSIS

#include <sys/malloc.h>

#define MALLOC(space, cast, size, type, flags)

PARAMETERS

space  Pointer to the kernel memory allocated.
cast   Type of the pointer space to be used as a cast in the macro.
size   The number of bytes (size) of kernel memory to allocate.
type   The memory allocation pool type.
flags  Flag to indicate the caller cannot block and wait for kernel memory availability.

DESCRIPTION

The MALLOC() and FREE() kernel macros are deprecated interfaces and may be obsoleted in a future release of HP-UX. Use the kernel functions kmalloc() and kfree() in place of MALLOC() and FREE().

The MALLOC() kernel macro allocates size bytes of kernel memory from the pool type specified. The memory allocated will always be contiguous in the virtual address space, but may be discontiguous in the physical address space. If size is greater than PAGESIZE, the physical pages allocated will probably not be physically contiguous.

Drivers should allocate memory from pool type M_IOSYS or M_DMA. The M_IOSYS pool is for general I/O purposes; the M_DMA pool is for DMA purposes.

The flags parameter may optionally have the M_NOWAIT flag set. If M_NOWAIT is set and no memory is available from the requested pool type, the functions will return NULL. Without M_NOWAIT set, the caller can be blocked and made to wait for memory to become available.
The \texttt{M\_NOWAIT} flag must be set if \texttt{MALLOC()} is:

- called in an interrupt context, or
- called while holding a spinlock

If \texttt{M\_NOWAIT} is set, the caller must be prepared to handle the case where no kernel memory has been allocated.

\texttt{M\_WAITOK} should be passed as the \textit{flags} parameter if the caller is able to block and wait for memory to become available.

**RETURN VALUES**

\texttt{MALLOC()} returns the following values to the pointer \textit{space}:

- \texttt{NULL} Virtual address of kernel memory allocated.
- \texttt{NULL} No kernel memory has been allocated. \texttt{NULL} is returned only when the \texttt{M\_NOWAIT} flag is set; otherwise, the caller is blocked and waits until memory becomes available.

**CONSTRAINTS**

If the \texttt{M\_NOWAIT} flag is not set:

- the caller must not be in an interrupt context, and
- no spinlocks can be held

**EXAMPLES**

```c
my_stuct_t * my_addr;

MALLOC (my_addr, my_struct_t *, sizeof(my_struct_t),
        M_IOSYS, M_NOWAIT);

if (!my_addr) {
    /* Error! No kernel memory currently available. */
}
```

SEE ALSO

FREE(KER2), kfree(KER2), kmalloc(KER2)
NAME

map_mem_to_host(KER2) – Map physical bus address to host virtual space

SYNOPSIS

#include <sys/wsio.h>

caddr_t map_mem_to_host (struct isc_table_type *isc,
    caddr_t phys_addr, size_t size);

PARAMETERS

isc ISD pointer that corresponds to the interface card
    associated with this memory.
phys_addr Base physical address for a range of memory space on
    an I/O bus.
size Size of the memory range in bytes.

DESCRIPTION

The kernel function map_mem_to_host() maps physical bus address to a
host virtual address for cards with memory ranges. Hardware disables
 caching for all EISA data accesses, including EISA memory space.
map_mem_to_host() will return NULL if a request overlaps an existing
mapping.

RETURN VALUES

map_mem_to_host() function returns the following values:

<>NULL Successful completion. The value is the host virtual
    address for accessing the space specified by the
    parameters.

NULL Error.
SEE ALSO

unmap_mem_from_host(KER2), kernel_iomap(KER2),
kernel_iounmap(KER2)
NAME

`minor(KER2)` - Extract the minor number from a device number.

SYNOPSIS

```c
#include <sys/sysmacros.h>
#define minor(x) ((long)(((x)&0xfffff))
```

PARAMETERS

`x`  
A `dev_t` device number.

DESCRIPTION

`minor()` is a macro that returns the minor number from a device number.

CONSTRAINTS

SEE ALSO

`major(KER2)`
NAME

minphys (KER2) - Limit the b_bcount field in a buf structure to the value MAXPHYS

SYNOPSIS

#include <sys/buf.h>

void minphys (struct buf *bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The minphys() kernel function compares bp->b_bcount against the value MAXPHYS defined in <sys/param.h>. If bp->b_bcount is greater than MAXPHYS, then bp->b_bcount is changed to MAXPHYS.

minphys() is passed as the mincnt parameter to physio(). In this way, physio() can break a large data transfer into multiple smaller transfers, each no greater than MAXPHYS bytes in length.

RETURN VALUES

None.

EXAMPLES

int
mydriver_read(dev_t dev, struct uio * uio)
{
    return physio(mydriver_strategy, NULL, dev, B_READ,
                   minphys, uio);
}

CONSTRAINTS
SEE ALSO

physio(KER2), driver_minphys(WSIO_DRV)
NAME

msg_printf(KER2) - Write diagnostic information to the kernel message buffer.

SYNOPSIS

```c
#include <sys/kern_svc.h>

int msg_printf (const char * format, ...);
```

PARAMETERS

format  A set of printing characters and limited conversion specifications, as defined in printf(3S).

DESCRIPTION

The msg_printf() kernel function is a scaled down version of the C library printf() routine (see printf(3S)).

msg_printf() writes diagnostic information to the kernel message buffer msgbuf only and not to the console.

The kernel msg_printf() routine can accept the following formats:

- **Format**  **Printed Result**
  - %%  %; no argument
  - %b  Characters from string argument; can include backslash-escape codes
  - %c  Character from integer argument
  - %d  Signed decimal from integer argument
  - %lx  The argument x is long integer; x is one of x, d, u, or o
  - %o  Octal from integer argument
  - %s  Characters from string argument
  - %u  Unsigned decimal from integer argument
  - %x  Hexadecimal from integer argument

These formats are the same as in printf(3S). Other formats specified in
printf(3S) are not supported.

RETURN VALUES

msg_printf() returns the length of the formatted string.

CONSTRAINTS

SEE ALSO

printf(KER2), printf(3S), sprintf(KER2), sprintf(3S)
NAME

owns_spinlock(KER2) – Test whether a spinlock is owned by the processor.

SYNOPSIS

#include <sys/spinlock.h>

int owns_spinlock (lock_t * lock);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The owns_spinlock() kernel function tests whether a spinlock, pointed to by lock, is owned (locked) by the processor.

RETURN VALUES

owns_spinlock() returns the following values:

1 The processor owns the spinlock.

0 The processor does not own the spinlock. It may be owned by another processor, or not owned by any processor.

CONSTRAINTS
EXAMPLE

```c
int got_spinlock_here = 0;

/*
 * We may have already acquired the spinlock before getting here. Be sure to check that we do not own the spinlock before acquiring the lock, otherwise, we may spin forever.
 */
if (!owns_spinlock(mydrv_lock)) {
    spinlock(mydrv_lock);
    got_spinlock_here = 1;
}

/*
 * Do work under the protection of the spinlock.
 */
...

/*
 * Release the spinlock if it was acquired here.
 */
if (got_spinlock_here) {
    spinunlock(mydrv_lock);
}
```

SEE ALSO

alloc_spinlock(KER2), cspinlock(KER2), dealloc_spinlock(KER2), get_sleep_lock(KER2), spinlock(KER2), spinunlock(KER2)
NAME

`p_pgrp(KER2)` - Return the process group identifier for a process.

SYNOPSIS

```c
#include <h/proc_iface.h>

pid_t p_pgrp (proc_t * procp);
```

PARAMETERS

`procp`  
Pointer to a `proc_t` structure.

DESCRIPTION

The `p_pgrp()` kernel function returns the process group identifier for a process.

The `procp` parameter is a pointer to a `proc_t` structure. The pointer for the current process is contained in `u.u_procp`.

RETURN VALUES

The `p_pgrp()` kernel function returns the process group identifier for a process.

CONSTRAINTS

EXAMPLES

```c
#include <sys/user.h>
#include <sys/signal.h>
#include <h/proc_iface.h>

/*
 * Signal all processes in the current process group.
 */
gsignal(p_pgrp(u.u_procp),SIGIO);
```
SEE ALSO

gsignal(KER2)
NAME

panic(KER2) - Soft-crash the operating system

SYNOPSIS

#include <sys/kern_svcs.h>

void panic (char * str);

PARAMETERS

str Pointer to a character string message. No format capability.

DESCRIPTION

The panic() kernel function prints str to the system console and halts the system.

panic() prints the processor status register, the program counter register, the trap type on processor exceptions, and part of the kernel stack.

RETURN VALUES

panic() does not return.

CONSTRAINTS

WARNINGS

panic() halts the system and may cause file system damage. This call should only be used to flag catastrophic and unrecoverable failures.
NAME

physio(KER2) – Perform unbuffered, physical I/O data transfers.

SYNOPSIS

#include <sys/buf.h>

int physio (int (*strategy)(), struct buf *bp, int dev, int flag,
    void (*mincnt)(), struct uio *uiop);

PARAMETERS

strategy Pointer to the driver strategy function.
bp Pointer to a buf structure.
dev Device number.
flag Read/write flag: B_READ or B_WRITE.
mincnt Function that limits maximum transfer length.
uiop Pointer to uio structure.

DESCRIPTION

The physio() kernel function performs unbuffered, physical I/O data transfers. It accepts a user I/O request specified by a uio structure, prepares the data pages for I/O, builds the associated buf structure, calls the specified driver strategy function, and waits for the I/O to complete.

The strategy parameter is a pointer to the driver strategy function. physio() calls strategy to start the I/O transfer, then waits by calling biowait(). When the I/O transfer completes, strategy calls biodone() to awaken the waiting thread. strategy reports an error to physio() by setting B_ERROR in bp->b_flags and putting an errno value in bp->b_error. See errno values in <sys/errno.h>.

The bp parameter is a pointer to a buf structure allocated by the caller. If set to NULL, physio() allocates the buf structure for the caller. physio() deallocates the buf structure when the I/O request completes.

The dev parameter is the device number passed to the driver.
The `flag` parameter indicates the direction of the I/O transfer. `B_READ` is used to read data from the device into memory; `B_WRITE` is used to write data from memory to the device.

The `mincnt` parameter is a pointer to a function that limits the data transfer length. Typically, drivers pass the kernel supplied function `minphys()` as this parameter.

The `uiop` parameter is a pointer to the `uiocb` structure that is passed to the driver. The `uiocb` structure specifies the following:

- `uiocbiov` Pointer to an `ioworker` structure that contains the base address `iov_base` and transfer length `iov_len` of the I/O request.
- `uiocbiovcnt` Number of `ioworker` structures. If >1, `uiocbiov` points to an array of `ioworker` structures.
- `uiocboffset` Offset into device.
- `uiocbsseg` Type of memory segment to transfer. If set to `UIOSEG_USER`, `physio()` must be called in the user context.
- `uiocbrezid` Number of bytes of data remaining to be transferred.

For each `ioworker` structure, `physio()` performs the following:

1. Validate the user has appropriate access permissions for the data pages specified by `ioworker`.
2. Set up the `buf` structure with the following information:
   - `b_dev` is set to the device number.
   - `b_error` is set to zero.
   - `b_flags` is set with `B_BUSY`, `B_PHYS` and `B_RAW`. If the `flag` parameter is `B_WRITE`, `B_WRITE` is set.
   - `b_un.b_addr` is set to `iov_base`.
   - `b_bcount` is set to `iov_len`. This value can be adjusted by `mincnt`.
   - `b_blkno` is set to the `DEV_BSIZE` block number corresponding to `uiocboffset`.
3. Call `mincnt` to adjust the transfer length, if too large. If adjusted, `physio()` will make multiple calls to `strategy` until all the data specified by `iov_len` has been transformed (or an error occurs).
4. If `uio_seg` is not `UIOSEG_KERNEL`, lock down the data pages to be transferred.

5. If `uio_seg` is not `UIOSEG_KERNEL` and the driver has set `C_MAP_BUFFER_TO_KERNEL` in the `d_flags` field of its `drv_ops_t` structure, `physio()` maps the user buffer into kernel space. This additional overhead is performed for legacy drivers that do not know how to access user space in their strategy function.

6. Calls strategy passing the `buf` structure pointer `bp` as a parameter, then wait for the I/O request to complete by calling `biowait()`.

7. The driver calls `biodone()` when the I/O request completes to awaken the thread waiting in `biowait()`.

8. After the I/O request has completed, data pages that have been locked down are unlocked.

9. The following fields in the `uio` structure are updated:
   
   - `uio_resid` is decreased by the transfer length
   - `uio_offset` is increased by the transfer length

   After all `iovec` structures have been processed, the `buf` structure is released and `physio()` returns to the caller.

**RETURN VALUES**

`physio()` returns the following values:

- `0` Successful completion.
- `<>0` Error.

**CONSTRAINTS**

Must not be called in an interrupt context.

Must not be called while holding a spinlock.

**EXAMPLES**

```c
int mydriver_read(dev_t dev, struct uio * uio)
{
    return physio(mydriver_strategy, NULL, dev, B_READ,
```
SEE ALSO

biodone(KER2), biowait(KER2), buf(KER4), minphys(KER2),
uio(KER4)
NAME
printf(KER2) - Kernel print routine

SYNOPSIS

#include <sys/kern_svcs.h>

int printf (const char * fmt, ...);

PARAMETERS

fmt A set of printing characters and limited conversion specifications, as defined in printf(3S).
...
An argument corresponding to a format conversion specification, as defined in printf(3S).

DESCRIPTION

The printf() kernel function is a scaled down version of the C library printf() routine (see printf(3S)). printf() writes diagnostic information to the console and into the kernel message buffer msgbuf.

The kernel printf(), routine can accept the following formats:

- Format Printed Result
  %- %; no argument
  %b Characters from string argument; can include backslash-escape codes
  %c Character from integer argument
  %d Signed decimal from integer argument
  %lx The argument x is long integer; x is one of x, d, u, or o
  %o Octal from integer argument
  %s Characters from string argument
  %u Unsigned decimal from integer argument
Hexadecimal from integer argument

These formats are the same as in `printf(3S)`. Other formats specified in `printf(3S)` are not supported.

The kernel `printf()` routine is buffered. Therefore, console error messages may not be synchronous with the current kernel state.

**RETURN VALUES**

`printf()` returns the length of the formatted string.

**CONSTRAINTS**

**SEE ALSO**

`sprintf(KER2), msg_printf(KER2), printf(3S)`
Kernel Reference Pages
Functions and Structures

NAME

privlbcopy(KER2) – Copy data from a source buffer to a destination buffer using
global virtual addresses.

SYNOPSIS

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

int privlbcopy (space_t from_sid, caddr_t from_addr,
               space_t to_sid, caddr_t to_addr, size_t n);
```

PARAMETERS

- **from_sid**: Space ID of source buffer.
- **from_addr**: Address of source buffer.
- **to_sid**: Space ID of destination buffer.
- **to_addr**: Address of destination buffer.
- **n**: Number of bytes to copy.

DESCRIPTION

The `privlbcopy()` kernel function copies data from a source buffer to a
destination buffer using global virtual addresses. Protection is disabled
to allow copies from/to buffers that are not owned by the current process.

The `from_sid` and `from_addr` parameters are the space ID and address,
respectively, of the source buffer.

The `to_sid` and `to_addr` parameters are the space ID and address,
respectively, of the destination buffer.

The space ID (SID) and address are combined to form the global virtual
address of the buffer. If the address is in kernel space, then `lidsid()` can
be used to get the SID of the address.

Do not call `privlbcopy()` to copy data from/to user space without first
calling `physio()`. `physio()` validates access to the user buffer, locks the
 corresponding data pages in memory, puts the user SID and address in
the `b_spaddr` and `b_un.b_addr` fields of the `buf` structure. (`b_spaddr`
and `b_un.b_addr` will contain a kernel space ID and address if the driver
specifies C_MAP_BUFFER_TO_KERNEL in its \texttt{drv_ops_t} structure.)

**RETURN VALUES**

\texttt{privlbcopy()} returns the following values:

0 \hspace{1em} Successful completion.

<>0 \hspace{1em} Error.

**CONSTRAINTS**

**EXAMPLES**

\begin{verbatim}
/*
 * Copy data from a kernel buffer to a user buffer that
 * has been validated and locked by physio().
 */
(void)privlbcopy(ldsid(kern_addr), kern_addr,
    bp->b_spaddr, bp->b_un.b_addr, bp->b_bcount);
\end{verbatim}

**WARNINGS**

\texttt{privlbcopy()} must not be used for transfers between memory and I/O space. The underlying routines make choices of the optimal transfer code, which will probably not be supported on either the I/O bus adapter or the attached interface card. Failure to heed this warning may result in data corruption, an HPMC (High Priority Machine Check), or a call to \texttt{panic()}.

**SEE ALSO**

\texttt{bcopy(KER2), copyin(KER2), copyout(KER2), ldsid(KER2)}
\texttt{physio(KER2)}
NAME
psignal(KER2) - Send the specified signal to a process

SYNOPSIS
#include <h/proc_iface.h>
void psignal (proc_t * procp, int sig);

PARAMETERS
procp Pointer to a proc_t structure.
sig Signal number.

DESCRIPTION
The psignal() kernel function sends the specified sig to the process specified by procp.

The procp parameter is a pointer to a proc_t structure. The pointer for the current process is contained in u.u_procp.

Signal numbers are defined in <sys/signal.h>.

psignal() checks the signal mask of the process to determine if the specified signal is being blocked, ignored, or caught. If a process is blocking the signal, the signal is recorded as pending so the signal is not lost. If a process is ignoring the signal, the signal is not sent. If the process is catching the signal, that process is put on the run queue. For example, if the process has called sleep() with priority greater than PZERO and PCATCH set, psignal() will awaken the sleeping process.

RETURN VALUES
None.

CONSTRAINTS
EXAMPLES

#include <sys/user.h>
#include <sys/signal.h>
#include <h/proc_iface.h>

/*
 * In the user context, save the u_procp.
 */
proc_t * my_procp = u.u_procp;
...

/*
 * In the ISR, signal the process.
 */
psignal(my_procp, SIGIO);

SEE ALSO

gsignal(KER2)
NAME
putc(KER2) – Manipulate characters on a clist

SYNOPSIS

#include <sys/clist.h>

int putc(int ch, struct clist *list);

PARAMETERS

ch Character to place on the clist.
list Pointer to a clist.

DESCRIPTION

The putc() kernel function puts the character ch on the clist list.

RETURN VALUES

putc() returns the following values:

0 Successful completion.
-1 Error (probably exhausted the available list structures).

CONSTRAINTS

EXAMPLES

void catq( from, to )
struct clist * from;
struct clist * to;

/* concatenate characters from one clist to another */
/* expects putc to succeed */
{
    int c;
while ((c = getc(from)) >= 0)
    if(putc(c, to))
}

SEE ALSO

getc(KER2), getcb(KER2), getcf(KER2), putcb(KER2), putcf(KER2)
NAME

putcb(KER2) – Manipulate cblocks on a dist

SYNOPSIS

#include <sys/clist.h>

void putcb (struct cblock * cb, struct clist * list);

PARAMETERS

    cb      Pointer to a cblock.
    list    Pointer to a dist.

DESCRIPTION

putcb() adds the cblock pointed to by cb onto the dist pointed to by list.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

getcb(KER2), getc(KER2), getcf(KER2), putc(KER2), putcf(KER2)
NAME

putcf(KER2) - Manipulate a cblock on the cblock free list

SYNOPSIS

#include <sys/clist.h>

void putcf (struct cblock *cb);

PARAMETERS

cb Pointer to a cblock structure.

DESCRIPTION

putcf() returns the cblock pointed to by cb to the free list.

RETURN VALUES

None.

CONSTRAINTS

EXAMPLES

static int
mydev_open(dev_t dev, int flag)
{
struct cblock *cp;
struct mydevice dv;

  /* Flush out the device buffer */
  while ((cp = getcb(&dv->queue)) != NULL)
    putcf(cp);
  ...
}
SEE ALSO

getcf(KER2), getc(KER2), getcb(KER2), putc(KER2), putcb(KER2)
NAME

selwakeup(KER2) – Wake up a kernel thread sleeping on a select condition

SYNOPSIS

void selwakeup (struct kthread * thread, int collision);

PARAMETERS

thread Pointer to the kernel thread to be awakened.
collision Indicates another kernel thread is sleeping on a select condition.

DESCRIPTION

The selwakeup() kernel function wakes up a kernel thread sleeping on a select condition. If collision is zero, only the kernel thread pointed to by thread is awakened. If collision is non-zero, all kernel threads sleeping on a select condition are awakened.

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock of order >= SCHED_LOCK_ORDER.

EXAMPLE

#define MYSEL_COLL 1    /* my select collision flag */

static int
mydev_select(dev_t dev, int rw)
{
    struct kthread * kthread;
    ...
    /*
    * Check if another kernel thread is already
    */

* sleeping on a select condition with the driver. */
kthread = myselstruct->thread;
if (kthread &
    (kt_wchan(kthread) == (caddr_t)selwait)) {
    /*
     * There is going to be a collision of multiple
     * threads sleeping on a select condition.
     */
    myselstruct->selflag |= MYSEL_COLL;
} else {
    /*
     * Save the kernel thread pointer of the first
     * (and only?) thread to sleep on a select condition.
     */
    myselstruct->thread = u.u_kthreadp;
}
...

static void
myselwakeup(void)
{
    selwakeup(myselstruct->thread,
              myselstruct->selflag & MYSEL_COLL);
    myselstruct->thread = NULL;
    myselstruct->selflag &= ~MYSEL_COLL;
}
NAME

sleep(KER2) - Sleep on a channel

SYNOPSIS

#include <sys/kern_svc.h>
#include <sys/param.h>

int sleep (caddr_t chan, int pri);

PARAMETERS

chan Channel (i.e., kernel address) to sleep on.
pri Sleep priority.

DESCRIPTION

The sleep() kernel function sleeps on a channel chan and waits for a corresponding call to wakeup(). When awakened, the sleeping thread is scheduled at priority pri.

The chan parameter is typically a unique kernel address, such as a pointer to a private data area. This provides a one-to-one correspondence between wakeup events and chan. chan must not be zero.

The pri parameter specifies a value related to the system priority level of the sleeping thread. Block I/O drivers should set pri to the value PRIBIO, which is below the signal threshold value PZERO. If pri is set to a value greater than PZERO and the PCATCH flag is set (ORed with pri), the sleeping thread may be interrupted and awakened by a signal. If pri is greater than PZERO and PCATCH is not set, a signal handler may be invoked, and the call to sleep() may never return to the caller.

If more than one thread sleeps on chan, there is a race as to which thread wakes up first. All threads waiting for an event are awakened together. Each thread that returns from its call to sleep() should verify it is to handle the event; threads not handling the event may need to call sleep() again.

Prior to calling sleep(), a call to get_sleep_lock() may be required to protect against a race condition with wakeup(). The typical sequence executed by drivers is to:
• Acquire the sleep lock.
• Start an asynchronous activity
• Sleep and wait for the asynchronous activity to complete.

The sleep lock is released by `sleep()` after the thread has been put on the sleep queue. This ensures that the thread will not miss the corresponding `wakeup()` which also acquires the same sleep lock.

Acquiring the sleep lock is not required if there is no race condition with `wakeup()`. For example, consider a driver set up to call `wakeup()` at regular intervals in a callout function specified by `timeout()`. A call to `sleep()` may miss a corresponding call to `wakeup()`, but the thread will be awakened by the next call to `wakeup()`, which occurs at regular intervals.

### RETURN VALUES

`sleep()` returns the following values:

- **0** Awakened by a corresponding call to `wakeup()`.
- **1** Awakened by a signal if the `PCATCH` flag is set.

### CONSTRAINTS

Must not be called in an interrupt context.

Must not be called while holding a spinlock.

### EXAMPLE

```c
/*
 * The sleeping thread waits on a channel by first calling
 * `get_sleep_lock()`, then it starts an asynchronous activity
 * and finally calls `sleep()` which releases the sleep lock.
 */
(void)get_sleep_lock(wait_chan);
start_async_activity();
(void)sleep(wait_chan, PRIBIO);
... 
/*
 * When the asynchronous activity completes (typically in an
 * interrupt service routine), the sleeping thread is awakened.
 */
```
(void) wakeup(wait_chan);

**SEE ALSO**

get_sleep_lock(KER2), wakeup(KER2),
NAME

spinlock(KER2) - Acquire (lock) a spinlock

SYNOPSIS

#include <sys/spinlock.h>

void spinlock (lock_t * lock);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The spinlock() kernel function attempts to acquire (lock) the spinlock pointed to by lock. The processor spins and waits if the spinlock is currently locked.

Observe the following restrictions while holding a spinlock:

• Do not hold a spinlock for more than a few milliseconds.
• Do not acquire another spinlock of lower or equal lock order.
• Do not call an interface that can potentially block (i.e., sleep).

Spinlocks can be acquired while executing in an interrupt context as well as a user context. As such, they are able to synchronize the top and bottom halves of a driver.

RETURN VALUES

None
CONSTRAINTS

EXAMPLE

/*
 * Acquire the spinlock for my driver.
 */
spinlock(mydrv_lock);

/*
 * Manipulate driver data protected by the spinlock.
 * Do as little as possible here since external interrupts
 * to the processor are disabled while a spinlock is held.
 */
...

/*
 * Release the spinlock when done.
 */
spinunlock(mydrv_lock);

SEE ALSO

alloc_spinlock(KER2), cspinlock(KER2), dealloc_spinlock(KER2),
get_sleep_lock(KER2), owns_spinlock(KER2), spinunlock(KER2)
NAME

spinunlock(KER2) - Release (unlock) a spinlock.

SYNOPSIS

#include <sys/spinlock.h>

void spinunlock (lock_t *lock)

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting -- they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

The spinunlock() kernel function releases (unlocks) the spinlock pointed to by lock.

RETURN VALUES

None

CONSTRAINTS

EXAMPLE

/*
 * Acquire the spinlock for my driver.
 */
spinlock(mydrv_lock);

/*
 * Manipulate driver data protected by the spinlock.
 */
* Do as little as possible here since external interrupts
* to the processor are disabled while a spinlock is held.
*/
...

/*
 * Release the spinlock when done.
 */
spinunlock(mydrv_lock);

SEE ALSO
alloc_spinlock(KER2), cspinlock(KER2), dealloc_spinlock(KER2),
get_sleep_lock(KER2), owns_spinlock(KER2), spinlock(KER2)
NAME

sprintf(KER2) - Kernel version of sprintf().

SYNOPSIS

#include <sys/kern_svcs.h>

int sprintf (char *str, int len, const char *fmt, ...);

PARAMETERS

str Address of buffer to hold the formatted string.
len Length of the formatted string in bytes.
fmt A set of printing characters and limited conversion specifications, as defined in sprintf(3S).
... An argument corresponding to a format conversion specification, as defined in sprintf(3S).

DESCRIPTION

The sprintf() kernel function is a scaled down version of the C library sprintf() routine (see sprintf(3S)). However, the kernel version adds len as the second parameter.

sprintf() writes information to the str array.

The kernel sprintf(), routine can accept the following formats:

- Format Printed Result
  %% ; no argument
  %b Characters from string argument; can include backslash-escape codes
  %c Character from integer argument
  %d Signed decimal from integer argument
  %lx The argument x is long integer; x is one of x, d, u, or o.
  %o Octal from integer argument
%s  Characters from string argument
%u  Unsigned decimal from integer argument
%x  Hexadecimal from integer argument

These formats are the same as in sprint(3S). Other formats specified in sprintf(3S) are not supported.

RETURN VALUES

sprintf() returns the length of formatted string.

CONSTRAINTS

SEE ALSO

msg_printf(KER2), printf(KER2), sprintf(3S)
NAME

strcat(KER2) – Concatenate two strings

SYNOPSIS

#include <sys/kern_svc.h>

char * strcat (char * buf, const char * str);

PARAMETERS

buf Pointer to a string.
str Pointer to a string.

DESCRIPTION

The strcat() kernel function appends string buf to string str.

RETURN VALUES

strcat() returns buf.

CONSTRAINTS

SEE ALSO

strcpy(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME

strcmp(KER2) – Compare two strings

SYNOPSIS

#include <sys/kern_svcs.h>

int strcmp (char * s1, char * s2);

PARAMETERS

s1          Pointer to a character string.
s2          Pointer to a character string.

DESCRIPTION

The strcmp() kernel function compares two null-terminated strings. If
the strings are the same, zero is returned. If the strings are different, the
integer value of (*s1-*s2) is returned.

RETURN VALUES

strcmp() returns the following values:

0           The strings are identical.
<>0          The strings are different.

CONSTRAINTS

EXAMPLES

char *
scsi_decode_opcode(dev_type, op)
int dev_type;
int op;
{
   struct scsi_opcode *entry = scsi_opcode[op];
   int dev_type_bit = 1 << dev_type;
   int i;
static int initialized;

/*
 ** Do a sanity check on the scsi_opcode table.
 */
if (!initialized)
{
    i = strcmp(scsi_opcode[CMDread_element_status][0].op_str,
               "Read Element Status");
    if (i == 0)
        initialized = 1;
    else
        return "BadTable";
}

SEE ALSO

strcpy(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME
strcpy(KER2) – Copy the characters from one string to another string

SYNOPSIS

#include <sys/kern_svcs.h>

char * strcpy (char *s1, char *s2);

PARAMETERS

s1 Pointer to a string.
s2 Pointer to a string.

DESCRIPTION

The strcpy() kernel function copies the characters from the string pointed to by s2 to the string pointed to by s1, terminating at the first null character in s2. A pointer to the string s1 is returned. No bounds checking is done.

RETURN VALUES

strcpy() returns the string pointed to by s1.

CONSTRAINTS

EXAMPLES

{
  .
  .
  if ( slot_is_isa )
    strcpy(&desc[0],"ISA card ");
  else {
    strcpy(desc,"EISA card ");
    cvt_eisa_id_to_ascii(valid_card_id, id_str);
    /* id_str returns 8 bytes with last byte null */
    strcpy(&desc[strlen(desc)], id_str);
  }
Kernel Reference Pages
Functions and Structures

SEE ALSO

strcmp(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME

strlen(KER2) – Gets the number of nonnull bytes in a string

SYNOPSIS

#include <sys/kern_svc.h>

int strlen (char * s);

PARAMETERS

s Pointer to the string.

DESCRIPTION

The strlen() kernel function returns the number of bytes in the string, up to but not including the first null.

RETURN VALUES

strlen() returns the number of nonnull bytes in the string.

CONSTRAINTS

EXAMPLES

/*
 * Hex dump a chunk of data with header into the dmesg
 * buffer.
 */
msg_dump(label, buf, n)
char *label;
u_char *buf;
int n;
{
    int i, j, indent;

    msg_printf("\t%s:", label);
    indent = strlen(label) + 1;

    for (i = 0; i < n; i++)
        msg_printf("\x%02x", buf[i]);
    msg_printf("\x\n");

    for (j = 0; j < indent; j++)
        msg_printf("\t");
    }

    msg_printf("\n\n");
}

}
for (i = 0; i < n; i++)
{
    if ((i & 0xf) == 0 & i != 0)
    {
        msg_printf("\n\t");
        for (j = 0; j < indent; j++)
            msg_printf(" ");
        msg_printf(" %02x", buf[i]);
    }
    msg_printf("\n");
}

SEE ALSO

strcmp(KER2), strcpy(KER2), strncmp(KER2), strncpy(KER2)
NAME

`strn cmp(KER2)` - Compare the first n characters of two strings

SYNOPSIS

```c
#include <sys/kern_svcs.h>

int strncmp ( char *s1, char *s2, int n);
```

PARAMETERS

- `s1` Pointer to a string.
- `s2` Pointer to a string.
- `n` Number of bytes to compare.

DESCRIPTION

The `strncmp()` kernel function compares the first n characters of two null-terminated strings. If the strings are the same, zero is returned. If the strings are different, the integer value of (*s1 - *s2) is returned.

RETURN VALUES

`strncmp()` returns the following values:

- 0 The strings are equivalent.
- <>0 The strings are different.

CONSTRAINTS

EXAMPLES

```c

{
    .
    .
    if ((strncmp(utsname.machine, "9000/720", 8) != 0) &&
        (strncmp(utsname.machine, "9000/730", 8) != 0) &&
        (strncmp(utsname.machine, "9000/750", 8) != 0)) {
```

Chapter 2  141
/*  *
 * This isn’t a 720, 730 or 750. If there was an  *
 * audio card  *
 * it would have been found by pdc at boot time.  *
 */  
    return((struct audio_descriptor *) 0);  
  
  

SEE ALSO

strcmp(KER2), strlen(KER2), strncmp(KER2), strncpy(KER2)
NAME

`strncpy(KER2)` - Copy characters between strings

SYNOPSIS

```c
#include <sys/kern_svcs.h>

char * strncpy(char * s1, char * s2, int n);
```

PARAMETERS

- `s1` Pointer to a string.
- `s2` Pointer to a string.
- `n` Number of bytes to copy.

DESCRIPTION

The `strncpy()` kernel function copies the characters from the string pointed to by `s2` to the string pointed to by `s1`. Copying terminates at `n` bytes or after a NULL character which ever occurs first. No bounds checking is done.

RETURN VALUES

A pointer to the string `s1` is returned. `strncpy()` returns a pointer to the copied (`s1`) string.

CONSTRAINTS

EXAMPLES

```c
{
    .
    .
    /*
     ** Get the first four bytes of the inquiry data
     ** for dev_id and the vendor and product id for desc
     */
```
functions and structures

```c
strncpy(desc, iqr_data->inq2.vendor_id, 8);
strncpy(&desc[8], iqr_data->inq2.product_id, 16);
desc[24] = '\0';
```

See also

`string(3C), strcmp(KER2), strlen(KER2), strncmp(KER2), strcpy(KER2)`
NAME

\texttt{suser(KER2)} - Test if the current user is a superuser

SYNOPSIS

\begin{verbatim}
#include <sys/kern_svcs.h>

int suser (void);
\end{verbatim}

PARAMETERS

None.

DESCRIPTION

The \texttt{suser()} kernel function tests if the current user is a superuser.

RETURN VALUES

\texttt{suser()} returns the following values:

\begin{itemize}
  \item \texttt{0} \quad \text{The current user is a superuser.}
  \item \texttt{-1} \quad \text{The current user is not a superuser.}
\end{itemize}

CONSTRAINTS

Must be called in the user context.

SEE ALSO
NAME

sw_trigger(KER2) – Request a software trigger

SYNOPSIS

#include <sys/timeout.h>

void sw_trigger (struct sw_intloc *intloc, int (*proc)(), char *arg,
int level, int sublevel);

PARAMETERS

intloc A pointer to a sw_intloc structure to be added to the
queue of software triggers. The driver allocates the
structure, zero-filled. The sw_trigger() routine
initializes its fields.

proc The address of a routine to be called when the software
trigger is executed.

arg The argument to be passed to proc.

level The priority level of the software trigger.

The level value has the following restrictions:

• Your driver cannot set a software trigger higher
than your current processor priority level.

• You can not call sw_trigger() with level set to 7.

sublevel Currently, sublevels are not implemented. Drivers can
safely use 0 as the last argument.

DESCRIPTION

The software trigger mechanism provides software triggering of
interrupt service routines. The sw_trigger() routine arranges the
calling of a routine in interrupt context at a given priority level.

• Your timeout routine can set up a software trigger so that it defers its
timeout processing from level 5 to a lower level.

• Use a software trigger when your driver needs to acknowledge a
device's interrupt quickly, at a high level, but can do the rest of the interrupt processing less urgently, at a lower level.

- Software triggers provide a way for the top half of a driver to trigger the lower half to perform a specific function.

The kernel uses a linked list of structures to represent software triggers waiting to be serviced. The kernel checks this list each time it finishes servicing an interrupt. Elements of the list are `sw_intloc` structures, defined in `/usr/include/sys/timeout.h`.

When it checks the list, the kernel processes all requests for software triggers whose `level` is greater than the current interrupt level. The kernel processes pending requests in decreasing order of priority.

The `sw_trigger()` routine checks to see if the structure to which `intloc` points is already on the trigger queue. If it is, the kernel throws this request away, thus permitting only one pending request per `sw_intloc` structure. If your driver needs to have more than one software trigger pending, it must use separate `sw_intloc` structures.

**RETURN VALUES**

`sw_trigger()` is a void function.

**CONSTRAINTS**

**EXAMPLES**

The following fragment of a skeleton driver acknowledges an interrupt from a card at a high priority, and then uses a software trigger to defer the bulk of the interrupt processing to a lower priority.

```c
#include <sys/types.h>
#include <sys/timeout.h>
struct sw_intloc mycard_intloc;

mycard_isr()
{
  int reason;
  /* stop card from interrupting */
  mycard->control = ......
```
/* determine reason for interrupt and do *
 * any immediate interrupt processing *
*/
reason = ...; /* values from card regs */

/* set up sw_trigger() request to perform *
 * remainder of interrupt processing at *
 * a lower level *
*/
sw_trigger (<exc|&|mycard_intloc,mycard_isrII,reason,3,0);

return(0);
}

mycard_isrII( reason )
int reason; /* reason for interrupt, */
/* passed by mycard_isr() */
{
/* complete secondary interrupt processing */
switch(reason)
{
  case IOCOMPLETE:
    /* process I/O complete condition */
  case IOERROR:
    /* processing for I/O error */
    .
    .
    .
  }
}

SEE ALSO
NAME

timeout (KER2) - Execute a callout function after a specified length of time.

SYNOPSIS

```c
#include <sys/param.h>
#include <sys/callout.h>

struct callout * timeout(int (* func)(), caddr_t arg, int t);
```

PARAMETERS

- `func` Function to call when the time value `t` expires.
- `arg` Argument passed to the callout function `func`.
- `t` Time value in number of clock ticks.

DESCRIPTION

The `timeout()` kernel function executes the specified callout function `func` after `t` clock ticks have expired. Execution of `func` takes place in an interrupt context at priority level 2 where external interrupts to the processor are enabled.

The `func` parameter is a pointer to a function that takes one argument. Although the prototype declares the function to return an int value, the kernel does not make use of the return value.

The `arg` parameter is passed as the one argument to `func`.

The `t` parameter specifies the number of clock ticks to wait before calling `func`. To express time in seconds, multiply `t` by `HZ`, where `HZ` is defined as the number of clock ticks per second in `<sys/param.h>`.

The call to `timeout()` returns immediately without waiting for the time value `t` to expire. The timeout can be cancelled by making a corresponding call to `untimeout()`.

RETURN VALUES

- `timeout()` returns a pointer to a callout structure.
CONSTRAINTS

Must not be called while holding a spinlock of order \( \geq \) CALLOUT_LOCK_ORDER.

WARNINGS

Callout resources are not dynamically expandable. Each call to timeout() allocates a callout resource, and the resource is not released until the time value expires or the timeout is cancelled. The kernel may panic if no callout resources are available.

EXAMPLES

/*
 * Set a timeout to call my_timeout_func() passing my_arg
 * after 5 seconds have expired.
 */
(void)timeout(my_timeout_func, my_arg, 5*HZ);
...

static int
my_timeout_func(caddr_t arg)
{
    ...
}

SEE ALSO

Ktimeout(KER2), untimout(KER2)
NAME

uio(KER4) – Data descriptor for scatter/gather I/O requests.

SYNOPSIS

#include <sys/uio.h>

DESCRIPTION

The uio structure specifies an I/O request that can be fragmented into multiple buffers (scatter/gather I/O). uio contains a pointer to an array of iovec structures, each specifying the base address and length of a buffer. Buffers may be in either user space or kernel space.

For read() and write() system calls, the kernel allocates and fills out a uio structure pointing to a single iovec structure. The uio structure for readv() and writev() system calls points to an array of iovec structures. The uio structure is passed to the driver_read() or driver_write() entry point of the target driver.

Drivers should not access fields in the uio structure directly. The kernel function physio() is called to perform raw, unbuffered I/O and uiomove() is called to copy the specified buffer(s) to/from a kernel buffer. Fields in the uio structure are updated by physio() and uiomove().

STRUCTURE MEMBERS

The uio structure is defined in <sys/uio.h>. It has the following fields.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct iovec *</td>
<td>uio_iov</td>
</tr>
<tr>
<td>size_t</td>
<td>uio_iovcnt</td>
</tr>
<tr>
<td>off_t</td>
<td>uio_offset</td>
</tr>
<tr>
<td>uint32_t</td>
<td>uio_seg</td>
</tr>
<tr>
<td>long</td>
<td>uio_resid</td>
</tr>
<tr>
<td>uint32_t</td>
<td>uio_fpflags</td>
</tr>
</tbody>
</table>
uio_iov  Pointer to an array of iovec structures. Each iovec structure contains the base address and length of a buffer.

uio_iovcnt  Number of entries in the array of iovec structures.

uio_offset  Starting logical byte address on the device where the data transfer is to occur. Applicability of this field to the driver is device dependent. uio_offset is usually applicable only to devices capable of seeking.

uio_seg  Indicates the addresses of the buffers are in user space (UIOSEG_USER) or kernel space (UIOSEG_KERNEL).

uio_resid  Number of bytes remaining to be transferred. Initially, this equals the sum of all the buffer lengths.

uio_fpflags  Flags that indicate whether the driver should not wait:

  FNDELAY  The driver should not wait if the requested data transfer can not be done immediately. The request should be terminated without returning an error code.

  FNBLOCK  The driver should not wait if the requested data transfer can not be done immediately. The request should be terminated and return EAGAIN as the error code.

CONSTRAINTS

SEE ALSO

iovec(KER4), open(2), physio(KER2), read(2), readv(2),
uiomove(KER2), write(2), writev(2)
NAME

uiomove(KER2) - Copy data between memory in kernel space and the space specified by a uio structure.

SYNOPSIS

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

int uiomove(caddr_t addr, int n, int rw, struct uio * uiop);
```

PARAMETERS

- `addr` Address of kernel memory.
- `n` Number of bytes to copy
- `rw` Copy direction: UIO_READ, UIO_WRITE
- `uiop` Pointer to a uio structure.

DESCRIPTION

The `uiomove()` kernel function copies data between memory in kernel space and the space specified by the uio structure pointed to by `uiop`. `uiomove()` is typically used to copy data between user space and kernel space.

The `addr` parameter is the address of kernel memory to/from which data are to be copied. Kernel memory can be allocated by calling `kmalloc()` or `geteblk()`. Do not allocate kernel memory on the stack if the allocation size is greater than 128 bytes, as this can potentially lead to a stack overflow.

The `n` parameter is the number of bytes to copy. Typically, `n` is set to the value in `uiop->uio_resid`. If this value is greater than the size of allocated kernel memory, multiple calls to `uiomove()` with a smaller value of `n` will be required.

The `rw` parameter determines the direction of the copy: UIO_READ copies data from `addr`, UIO_WRITE copies data to `addr`.

The `uiop` parameter is a pointer to the uio structure. If the `uio_segflag` field in the uio structure is UIO_USERSPACE, user space is to be accessed and `uiomove()` must be called in the user context.
uiomove() can be called in any user context or in an interrupt context.

The uio_iov field in the uio structure points to an iovec structure. If uio_iovcnt is greater than 1, uio_iov points to an array of iovec structures. uiomove() walks through the array of iovec structures as it performs the copy.

If the copy is successfully completed, fields in the uio structure are updated as follows:

- uio_offset is increased by n.
- uio_resid is decreased by n.

For a single iovec structure (uio_iovcnt = 1), a successful completion will update the iovec as follows:

- uio_iov->iov_base is increased by n.
- uio_iov->iov_len is decreased by n.

For an array of iovec structures (uio_iovcnt > 1), a successful completion will increment each iov_base field appropriately and decrement each iov_len field appropriately.

**RETURN VALUES**

uiomove() returns the following values:

- 0 Successful completion.
- -1 Error.

**CONSTRAINTS**

**EXAMPLES**

```c
static int
mydriver_write(dev_t dev, struct uio * uiop)
{
    struct buf * bp;
    int count = uiop->uio_iov->iov_len;

    bp = geteblk(count);
```
(void)uiomove(bp->b_un.b_addr, count, UIO_WRITE, uio);
brese(bp);
...

SEE ALSO

brese(KER2), geteblk(KER2)
NAME

unmap_mem_from_host(KER2) - Unmap physical bus from host virtual space

SYNOPSIS

#include <sys/wsio.h>

int unmap_mem_from_host (struct isc_table_type *isc,
                          caddr_t virt_addr, size_t size);

PARAMETERS

isc The ISC pointer used in the previous map_mem_to_host() call.
virt_addr The virtual address previously obtained with a map_mem_to_host() call,
size The size of the memory space in bytes.

DESCRIPTION

The kernel routine unmap_mem_from_host() is used to unmap previously mapped virtual space. Hardware disables caching for all EISA data accesses, including EISA memory space.

RETURN VALUES

unmap_mem_from_host() returns an indeterminate value. Treat it as if it was a void function.

CONSTRAINTS

SEE ALSO

map_mem_to_host(KER2)
NAME
untimeout(KER2) - Cancel a previous timeout request

SYNOPSIS
#include <sys/callout.h>

int untimeout (int (* func)(), caddr_t arg);

PARAMETERS
func
The func parameter from the previous timeout request.
arg
The arg parameter from the previous request.

DESCRIPTION
The untimeout() kernel function cancels a previous timeout request
made by a call to timeout() or Ktimeout(). The func and arg
parameters are compared against the list of timeout requests waiting to
expire. If a match is found, the timeout request is cancelled by taking it
off the list and releasing allocated resources.

RETURN VALUES
untimeout() returns the following values:
≥ 0 Number of clock ticks remaining in the canceled
timeout request.
-1 Timeout request not found. This is a normal return if
the timeout request has already expired.

CONSTRAINTS
Must not be called while holding a spinlock of order >=
CALLOUT_LOCK_ORDER.

SEE ALSO
Ktimeout(KER2), timeout(KER2)
NAME

user_iomap(KER2) – Map physical bus space to currently running process.

SYNOPSIS

#include <sys/iomap.h>

caddr_t user_iomap(caddr_t vaddr, caddr_t phys_addr,
                   pgcnt_t count);

PARAMETERS

vaddr NULL or equivalent to physical address
phys_addr The start of the host physical address (not the bus
relative address) for an area of memory on an I/O bus
(EISA, for example).
count The size of the memory space in pages (4 Kbyte pages).

DESCRIPTION

The kernel routine user_iomap() is used to map physical I/O bus space
into the process address space of a currently running process. If this is
the first mapping of this space, the mapping will be done with PROT_URW
protection. If this is not the first mapping, this mapping will inherit the
protection set by the first mapping of the space.

Multiple calls to user_iomap() can be issued by a driver for the same
I/O address range for the same or other processes. At process exit time,
any I/O regions mapped through the use of user_iomap() will be
automatically unmapped. Note that if other drivers and/or user
processes have the same space mapped, then the I/O space itself will only
be unmapped upon the last kernel_iounmap() or user_iounmap() request.

RETURN VALUES

user_iomap() routine returns the following values:

<>NULL Successful completion. The value is the host virtual
    address for accessing the space specified by the
parameters.

NULL Error.

CONSTRAINTS

EXAMPLES

*  
** example maps 1 page of I/O space at 
** 0xf0000000 to driver and the currently 
** running process with user read/write 
** protection 
*/

#include <sys/mman.h>

{
    int sm;
    char byte;
    caddr_t mapped_addr;
    caddr_t user_mapped_addr;
    caddr_t phys_addr = 0xf0000000;
    pgcnt_t count = 1;

    /*
** n.b. must be in user context
** map the space to the driver
** kernel_iomap need not be done
** before user_iomap - they can
** be done in any order
*/
    mapped_addr = 
        kernel_iomap(NULL, phys_addr, count, PROT_URW);
    if (mapped_addr == NULL)
        goto map_failed; /* bailout code for error */

    /*
** map in the same space to the 
** currently running process 
** user_mapped_addr should be 
** directly returned to the process
user_mapped_addr = user_iomap(NULL, phys_addr, count);

}

WARNINGS

Hardware disables caching for all EISA data accesses, including EISA memory space.

In the case of EISA (not ISA) cards, the EISA IO space is mapped (prior to calling the driver attach routine) with PROT_KRW. This mapping is for one page (4 Kbytes). If you desire user access to EISA IO space you will need to first unmap isc->if_reg_ptr with a call to unmap_mem_from_host(), and then remap the space with kernel_iomap_public(). A similar situation exists for PCI memory space.

user_iomap() will return NULL if a request overlaps an existing mapping. If the request exactly overlays (same start address and size) an existing mapping, the mapping services will return the host virtual address requested.

SEE ALSO

unmap_mem_from_host(KER2), user_iomap(KER2),
kernel_iounmap(KER2), user_iounmap(KER2)
NAME

user_iounmap(KER2) – Unmap physical bus from user process virtual space

SYNOPSIS

#include <sys/iomap.h>

int user_iounmap (caddr_t vaddr, uint count);

PARAMETERS

vaddr The virtual address previously obtained with a
user_iomap() call,
count The size of the memory space in pages (4 KByte pages).

DESCRIPTION

The kernel routine user_iounmap() is used to unmap, from the user
process, previously mapped virtual space. If other drivers and/or user
processes have the same space mapped, then the I/O space will only be
unmapped upon the last kernel_iounmap() or user_iounmap() call.

RETURN VALUES

unmap_mem_from() routine returns the following values:

1 Error.
0 Success.

CONSTRAINTS

SEE ALSO

kernel_iomap(KER2), kernel_iomap_public(KER2),
kernel_iounmap(KER2), user_iomap(KER2)
NAME

VASSERT(KER2) - Test an assertion if OSDEBUG kernel

SYNOPSIS

#include <sys/debug.h>

VASSERT (expr);

PARAMETERS

expr  An expression that evaluates to true or false.

DESCRIPTION

The VASSERT() kernel function calls panic() if the kernel was compiled with OSDEBUG on and if the expression expr is false. Otherwise, it does nothing.

RETURN VALUES

None

CONSTRAINTS

SEE ALSO

panic(KER2)
NAME

wakeup(KER2) – Wake up all threads sleeping on a channel.

SYNOPSIS

#include <sys/kern_svcs.h>

int wakeup(caddr_t chan);

PARAMETERS

chan       Channel passed in the corresponding call to sleep().

DESCRIPTION

The wakeup() kernel function wakes up all threads sleeping on a chan channel. Each thread sleeping on chan is scheduled with the priority specified when the thread called sleep().

The chan parameter is typically a unique kernel address, such as a pointer to a private data area. This provides a one-to-one correspondence between wakeup events and chan. chan must not be zero.

If more than one thread sleeps on chan, there is a race as to which thread wakes up first. All threads waiting for an event are awakened together. Each thread that returns from its call to sleep() should verify it is to handle the event; threads that do not handle the event may need to call sleep() again.

RETURN VALUES

wakeup() returns the number of threads awakened.

CONSTRAINTS

Must not be called while holding a spinlock of order greater than SLEEP_Q_LOCK_ORDER.

EXAMPLES

/*
 * The sleeping thread waits on a channel by first calling

* get_sleep_lock(), then it starts an asynchronous activity
* and finally calls sleep() which releases the sleep lock.
*/
(void)get_sleep_lock(wait_chan);
start_async_activity();
(void)sleep(wait_chan, PRIBIO);
...
/*
* When the asynchronous activity completes (typically in an
* interrupt service routine), the sleeping thread is awakened.
*/
(void)wakeup(wait_chan);

SEE ALSO

get_sleep_lock(KER2) sleep(KER2)
Chapter 3 165

3 CDIO Reference Pages
This chapter contains reference pages for driver support routines that are external to all CDIOs.
Structure and Macro Commands
NAME

dma_sync(CDIO3) – Synchronize processor and device views of memory

SYNOPSIS

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

#define dma_sync (Addr_Type, Addr, Length, Hints)
```

PARAMETERS

<table>
<thead>
<tr>
<th>Addr-type</th>
<th>Space ID corresponding to Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>Virtual address (processor view) of memory object.</td>
</tr>
<tr>
<td>Length</td>
<td>Size of the memory object, in bytes, pointed to by addr.</td>
</tr>
<tr>
<td>Hints</td>
<td>Bitwise OR of hints that change the behavior of dma_sync(). If no hints are given, the call results in a SYNC instruction on noncoherent platforms and a SYNCDMA on coherent and semcoherent platforms. The defined hints are:</td>
</tr>
<tr>
<td></td>
<td>IO_ACCESSED Perform function only if the page has been accessed by a processor.</td>
</tr>
<tr>
<td></td>
<td>IO_MODIFIED Perform function only if the page has been modified by a processor.</td>
</tr>
<tr>
<td></td>
<td>IO_NO_SYNC Inhibit execution of SYNC or SYNCDMA instructions.</td>
</tr>
<tr>
<td></td>
<td>IO_PREFETCHED Perform function only if the processor prefetches data.</td>
</tr>
<tr>
<td></td>
<td>IO_READ Purge processor caches for inbound data on noncoherent systems.</td>
</tr>
<tr>
<td></td>
<td>IO_SYNC_FORCPU Same as IO_READ.</td>
</tr>
<tr>
<td></td>
<td>IO_SYNC_FORDEV Same as IO_WRITE.</td>
</tr>
<tr>
<td></td>
<td>IO_SYNC_MEM Synchronize processor caches with host memory: caches are flushed to memory when used with IO_WRITE (even on coherent platforms).</td>
</tr>
</tbody>
</table>
DESCRIPTION

The `dma_sync()` CDIO macro has been superseded by `dma_sync_IO()`. New drivers are encouraged to use `dma_sync_IO()` which provides the following benefits:

• **Eliminates the overhead of the SYNCDMA instruction on semicoherent platforms when the IO_READ_START hint is used.** `dma_sync()` does not recognize the IO_READ_START hint.

• **Eliminates the overhead of the SYNCDMA instruction on fully coherent platforms.** `dma_sync()` does not distinguish between coherent and semicoherent platforms and issues unnecessary SYNCDMA instructions on coherent platforms.

Legacy drivers call `dma_sync()` to synchronize the processor caches with DMA transactions mastered by their devices.

There are three cases to consider where drivers must call `dma_sync()`.

These cases are prior to starting a write transaction, prior to starting a read transaction and after completing a read transaction:

• **Prior to starting a write transaction:**
  
  For each buffer that is to be written out, the driver must call `dma_sync()` with the IO_WRITE hint set. On noncoherent platforms, this will cause the associated processor caches to be flushed. For all but the last buffer, the IO_NO_SYNC hint should also be set to reduce the performance penalty of synchronizing the cache flushes on noncoherent platforms.

• **Prior to starting a read transaction:**
  
  For each buffer that is to be read into, the driver must call `dma_sync()` with the IO_READ hint set. On noncoherent platforms, this will cause the associated processor caches to be purged. For all but the last buffers, the IO_NO_SYNC hint should also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms.

• **After completing a read transaction:**
  
  For each buffer that has been read into, the drive must call
The `dma_sync()` function with the `IO_READ` hint set. On noncoherent platforms, this will cause the associated processor caches to be purged of data that may have been prefetched. For all but the last buffer, the `IO_NO_SYNC` hint should also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms. On semicoherent platforms, the processor caches will be made to synchronize with the data read when the `IO_NO_SYNC` hint is not set.

**CONSTRAINTS**

**WARNINGS**

Do not use the `IO_READ_START` hint with `dma_sync()`. `IO_READ_START` is new to `dma_sync_IO()`.

**SEE ALSO**

`dma_sync_IO(CDIO3)`
NAME

dma_sync_IO (CDIO3) – Synchronize processor and device views of memory.

SYNOPSIS

#include <sys/dma.h>

#define dma_sync_IO (Addr_Type, Addr, Length, Hints)

PARAMETERS

Addr-type Space ID corresponding to Addr.
Addr Virtual address (processor view) of memory object.
Length Size of the memory object, in bytes, pointed to by addr.
Hints Bitwise OR of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of
da tact of hints that change the behavior of

IO_ACCESSSED Perform function only if the page has been accessed by a processor.

IO_MODIFIED Perform function only if the page has been modified by a processor.

IO_NO_SYNC Inhibit execution of SYNC or SYNDMA instructions.

IO_PREFETCHED Perform function only if the processor prefetches data.

IO_READ Purge processor caches for inbound data on noncoherent platforms. Done after completing the DMA data transfer.

IO_READ_START Purge processor caches for inbound data on noncoherent platforms and inhibit the SYNDMA instruction on semicoherent platforms. Done prior to starting the DMA data transfer.

IO_SYNC_FORCPU Same as IO_READ.
**IO_SYNC_FORDEV**  Same as **IO_WRITE**.

**IO_SYNC_MEM**  Synchronize processor caches with host memory: caches are flushed to memory when used with **IO_WRITE** (even on coherent platforms).

**IO_WRITE**  Flush processor caches for outbound data on noncoherent platforms.

**DESCRIPTION**

Drivers call `dma_sync_IO()` to synchronize the processor caches with DMA transactions mastered by their devices. `dma_sync_IO()` is sensitive to the underlying coherency of the platform. If the platform is coherent, `dma_sync_IO()` does nothing; the hardware provides the coherency functionality. If the platform is semicoherent, `dma_sync_IO()` handles the special case where the processor caches must be synchronized with data that have been read into host memory. If the platform is noncoherent, `dma_sync_IO()` flushes (or purges) and synchronizes the processor caches to maintain a consistent view of memory between processors and devices.

There are three cases to consider where drivers must call `dma_sync_IO()`. These cases are prior to starting a write transaction, prior to starting a read transaction and after completing a read transaction.

- **Prior to starting a write transaction:**
  
  For each buffer that is to be written out, the driver must call `dma_sync_IO()` with the **IO_WRITE** hint set. On noncoherent platforms, this will cause the associated processor caches to be flushed. For all but the last buffer, the **IO_NO_SYNC** hint should also be set to reduce the performance penalty of synchronizing the cache flushes on noncoherent platforms.

- **Prior to starting a read transaction:**
  
  For each buffer that is to be read into, the driver must call `dma_sync_IO()` with the **IO_READ_START** hint set. On noncoherent platforms, this will cause the associated processor caches to be purged. For all but the last buffers, the **IO_NO_SYNC** hint should also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms.
• After completing a read transaction:

For each buffer that has been read into, the drive must call 
dma_sync_IO() with the IO_READ hint set. On noncoherent 
platforms, this will cause the associated processor caches to be purged 
of data that may have been prefetched. For all but the last buffer, the 
IO_NO_SYNC hint should also be set to reduce the performance 
penalty of synchronizing the cache purges on noncoherent platforms. 
On semicoherent platforms, the processor caches will be made to 
synchronize with the data read when the IO_NO_SYNC hint is not set.

CONSTRAINTS

SEE ALSO

dma_sync(CDIO3)
NAME
drv_info(CDIO4) - Driver information structure

SYNOPSIS
#include <sys/conf.h>

PARAMETERS
typedef struct drv_info
{
    char *name;    /* Name of driver */
    char *class;   /* Device class (see below)*/
    ubit32 flags; /* Device flags (see below)*/
    int b_major;  /* Block device major number */
    int c_major;  /* Character device major number */
    cdio_t *cdio; /* Drivers set this to NULL */
    void *gio_private;
    /* Drivers set this to NULL */
    void *cdio_private;
    /* Drivers set this to NULL */
} drv_info_t;

DESCRIPTION
All CDIOs use the driver-specific fields in the drv_info_t CDIO
structure type, defined in <sys/conf.h>, to describe certain parameters
of the driver. A drv_info_t structure must be statically allocated.

The relevant fields are described below. All other fields in a drv_info_t
should be NULL.

STRUCTURE MEMBERS

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Pointer to a string containing the name of the driver. This is the name you use in the system file (usually /stand/system) in the $DRIVER_INSTALL section of a master file in /usr/conf/master.d, and as the value for driver in the driver_install() function name. See config(1M) and master(4). The current</td>
</tr>
</tbody>
</table>
implementation of kernel functions that access name require that the string be less than 16 characters long.

*class*  
Pointer to a string containing the name of the class that the driver is in. Interface drivers typically use `ext_bus`. Device drivers use classes that describe the general type of device, e.g., `disk`, `tape`, `pseudo`, etc. For interface drivers, instances of a card are enumerated within each class as they are identified by the kernel at boot time.

*flags*  
The bitwise OR of flag values that describe the driver, taken from:

- `DRV_CHAR`  
  Character device driver.
- `DRV_BLOCK`  
  Block device driver.
- `DRV_PSEUDO`  
  Pseudo driver.
- `DRV_SCAN`  
  Driver supports bus scanning.
- `DRV_MP_SAFE`  
  Driver provides its own multiprocessing protection. This flag and `C_MGR_IS_MP` in `drv_ops_t` must be consistent or the kernel services will treat the driver as if it were not MPSAFE.
- `DRV_SAVE_CONF`  
  Save configuration information to `/etc/ioconfig`. This file retains potentially volatile information, such as dynamic major numbers and card instance numbers, across reboots.

*b_major*  
The major number if this is a block device. Set it to −1 for dynamic assignment or if it is not a block device.

*c_major*  
The major number if this is a character device. Set it to −1 for dynamic assignment or if it is not a character device.

---

**NOTE**  
The values you specify above for `b_major` and `c_major` override the values you enter in a master file in `/usr/conf/master.d` (see `master(4)`).
SEE ALSO

config(1M), driver_install(WSIO_DRV), drv_ops(CDIO4),
wsio_drv_info(WSIO4), master(4)
NAME

drv_ops(CDIO4) - Structure to specify driver entry points

SYNOPSIS

#include <sys/conf.h>

PARAMETERS

typedef struct drv_ops
{
    int (*d_open)();   /* block and character */
    int (*d_close)();   /* block and character */
    int (*d_strategy)(); /* block */
    int (*d_dump)(); /* NULL (obsolete) */
    int (*d_psize)(); /* block */
    int (*reserved0)(); /* NULL */
    int (*d_read)(); /* character */
    int (*d_write)(); /* character */
    int (*d_ioctl)(); /* character */
    int (*dSelect)(); /* character */
    int (*d_option1)(); /* NULL */
    pfilter_t *pfilter; /* block and character */
    int (*reserved1)(); /* NULL */
    int (*reserved2)(); /* NULL */
    int (*reserved3)(); /* NULL */
    int d_flags; /* block and character */
} drv_ops_t;

DESCRIPTION

The \texttt{drv_ops} CDIO structure type, defined in \texttt{<sys/conf.h>}, contains pointers to all driver entry points. A \texttt{drv_ops} structure must be statically allocated.

The relevant fields are described in the STRUCTURE MEMBERS section. All other fields in \texttt{drv_ops} should be \texttt{NULL}. Except as noted, entry points that don't apply to your driver or that your driver does not provide should be \texttt{NULL} (for example, \texttt{d_read()} has no meaning for a printer).
**STRUCTURE MEMBERS**

The Device Type column indicates whether the field applies to character-only, block-only, or both types of drivers.

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_open()</td>
<td>both</td>
<td>Pointer to your <code>driver_open()</code> routine, which enables a device for subsequent operations.</td>
</tr>
<tr>
<td>d_close()</td>
<td>both</td>
<td>Pointer to your <code>driver_close()</code> routine, which performs the tasks required when a device is closed.</td>
</tr>
<tr>
<td>d_strategy()</td>
<td>block</td>
<td>Pointer to your <code>driver_strategy()</code> routine, which queues I/O requests for either reading or writing.</td>
</tr>
<tr>
<td>d_psize()</td>
<td>block</td>
<td>Pointer to your <code>driver_psize()</code> routine. For a swapping device, it should return the size of the swap partition.</td>
</tr>
<tr>
<td>d_read()</td>
<td>character</td>
<td>Pointer to your <code>driver_read()</code> routine, which should return the requested data transferred from the device.</td>
</tr>
<tr>
<td>d_write()</td>
<td>character</td>
<td>Pointer to your <code>driver_write()</code> routine, which should write the requested data to the device.</td>
</tr>
<tr>
<td>d_ioctl()</td>
<td>character</td>
<td>Pointer to your <code>driver_ioctl()</code> routine, which sends control information to, or gets it from, a device.</td>
</tr>
</tbody>
</table>
The flag bit defines for `d_flags` are:

- **C_ALLCLOSES**: Force a call to `driver_close()` on every closing of the device. (The default action is to call the driver’s close routine only on the last close of the device.)

- **C_NODELAY**: Tell the kernel to not wait for a write request to complete on this device. The default action is to wait for
a write request to complete before returning control to the calling process.

C_MGR_IS_MP Identify the driver as safe for use in a multiprocessing environment. This flag and the DRV_MP_SAFE flag in drv_info_t must be consistent or the kernel services will treat the driver as if it were not MP SAFE.

C_MAP_BUFFER_TO_KERNEL Identify that the device driver needs physio() to remap a user buffer to kernel space prior to calling the driver strategy() routine. This flag also identifies that after the associated buf structure has been marked iodone, physio() will remap the buffer to user space.

SEE ALSO

driver_close(WSIO_DRV), driver_ioctl(WSIO_DRV),
driver_open(WSIO_DRV), driver_psize(WSIO_DRV),
driver_read(WSIO_DRV), driver_select(WSIO_DRV),
driver_strategy(WSIO_DRV), driver_write(WSIO_DRV),
wsio_drv_info(WSIO_DRV), drv_info(CDIO4), physio(KER2),
select(2)
NAME

init_map_context (CDIO3) - Macro to initialize mapping context structure

SYNOPSIS

#include <sys/dma.h>

#define init_map_context (io_map_t *map-cb)

PARAMETERS

map-cb Pointer to a map control block struct.

DESCRIPTION

The init_map_context() CDIO macro initializes the mapping context structure for use by wsio_map(). Note that the use of a context structure in a wsio_map() call overrides any alternate allocation scheme that may have been specified by wsio_set_attributes().

The context structure may be used to map a single object for a single I/O, or it may be used to map multiple objects for multiple I/Os. This feature uses fewer system resources. This feature is most useful for noninterleaving devices (see wsio_set_attributes(WSIO3)) when the driver will map several objects and retain those mappings for the life of the driver (semipermanent mappings or reused mappings).

In the case where a context is used for multiple I/Os, wsio_unmap() must not be called for ANY objects mapped with a particular context until ALL of the I/Os mapped with that context have completed. Failure to ensure that all I/Os have completed may result in data corruption. It is the programmer’s responsibility to ensure that all I/Os mapped with a particular context are complete prior to unmapping any of the I/Os.

The context variable map may be either a local or global variable. It is the responsibility of the programmer to provide synchronization of this memory object.

EXAMPLE

See example in wsio_map(WSIO3).
SEE ALSO

wsio_map(WSIO3), wsio_fastmap(WSIO3),
wsio_set_attributes(WSIO3), wsio_unmap(WSIO3)
NAME

install_driver (CDIO3) - Install a driver's header structure into the CDIO.

SYNOPSIS

#include <gio.h>

int install_driver (drv_info_t *drv_info, drv_ops_t *drv_ops);

PARAMETERS

drv_info Pointer to the driver's drv_info_t structure.
drv_ops Pointer to the driver's drv_ops_t structure.

DESCRIPTION

The install_driver() CDIO function installs a driver's header structure outside any specific CDIO, typically for pseudo drivers. WSIO drivers must call wsio_install_driver (WSIO3).

RETURN VALUES

install_driver() returns the following values:

0 Successful completion.

-1 Error. The major number specified for the driver is already in use. The following message is displayed on the system console and in the error-log file:

   install_driver: Install of driver drv-info->name failed.

CONSTRAINTS

SEE ALSO

wsio_install_driver (WSIO3)
Chapter 4  WSIO Reference Pages
This chapter contains reference pages describing routines and data structures used by drivers to communicate with the WSIO CDIO.
Functions, Macros and Structures
NAME

dma_cleanup(WSIO3) – Clean up from a DMA transfer

SYNOPSIS

#include <wsio/wsio.h>

int * dma_cleanup (struct isc_table_type * isc,  
                   struct dma_parms * dma_parms);

PARAMETERS

isc Pointer to an ISC structure.

dma_parms Pointer to a dma_parms structure.

DESCRIPTION

The dma_cleanup() WSIO function performs the required cleanup to recover from a DMA transfer.

See eisa_dma_cleanup(EISA3) for further details.

RETURN VALUES

dma_cleanup() returns an indeterminate value. Treat it as a void functions.

CONSTRAINTS

SEE ALSO

dma_setup(WSIO3)
NAME
dma_parms (CDIO4) – Dma information structure

SYNOPSIS
#include <sys/io.h>

PARAMETERS

struct dma_parms
{
    int channel;
    int dma_options;
    int flags;
    int key;
    int num entries;
    buflet_info_type * buflet_key;
    struct iovec * chain_ptr;
    int chain_count;
    int chain_index;
    int (*drv_routine)(caddr_t drv_arg);
    caddr_t drv_arg;
    int transfer_size;
    caddr_t_addr; /* host offset address */
    space_t spaddr; /* host space address */
    int count;
};

#include <sys/eisa.h>/* dma_options bits */

#define DMA_ISA 0x1
#define DMA_TYPEA 0x2
#define DMA_TYPEB 0x4
#define DMA_BURST 0x8
#define DMA_TYPEC DMA_BURST
#define DMA_DEMAND 0x10
#define DMA_SINGLE 0x20
#define DMA_BLOCK 0x40
#define DMA_CASCADE 0x80
#define DMA_8BYTE 0x100
#define DMA_16WORD 0x200
WSIO Reference Pages
Functions, Macros and Structures

#define DMA_16BYTE 0x400
#define DMA_32BYTE 0x800
#define DMA_READ 0x1000
#define DMA_WRITE 0x2000

/* bit definitions for flags field - bits cleared */
/* are defaults */
#define ADDR_CHAIN 0x1
/* addr is pointer to chain of I/O’s */
#define NO_CHECK 0x2
/* don’t perform error checking */
#define NO_ALLOC_CHAIN 0x4
/* allocate chain for addr/count’s */

/* errors returned by dma_setup */
#define UNSUPPORTED_FLAG -1
#define RESOURCE_UNAVAILABLE -2
#define BUF_ALIGN_ERROR -3
#define MEMORY_ALLOC_FAILED -4
#define TRANSFER_SIZE -5
#define INVALID_OPTIONS_FLAGS -6
#define ILLEGAL_CHANNEL -7

DESCRIPTION

Legacy interface drivers can use the dma_parms structure for setting up a
dma transfer. The kernel routines dma_setup() and dma_cleanup() use
a pointer to this structure as one of the parameters.

New interface drivers use the various WSIO mapping services in
conjunction with iovec(KER4) structures to set up bus master mapping.

SEE ALSO

dma_cleanup(WSIO3), dma_setup(WSIO3), wsio_map(WSIO3),
iovec(KER4)
NAME
dma_setup(WSIO3) – Set up and return information for a DMA transfer

SYNOPSIS

#include <wsio/wsio.h>

int * dma_setup (struct isc_table_type * isc,
                struct dma_parms * dma_parm);

PARAMETERS

isc             Pointer to an ISC structure.
dma_parms      Pointer to a dma_parms structure.

DESCRIPTION

The dma_setup() WSIO function performs the required setup and returns the necessary information for a DMA transfer.

RETURN VALUES

dma_setup() returns the following values:

0               Successful completion.
<>0             Error. Values depend upon the underlying CDIO function that is called.

CONSTRAINTS

SEE ALSO

dma_cleanup(WSIO3)
NAME

\texttt{driver\_addr\_probe(WSIO\_DRV)} - Provide an interface driver specific probing function.

SYNOPSIS

\begin{verbatim}
#include <sys/wsio.h>
#include <sys/ioparams.h>

driver_addr_probe(void * this_node, int (* dev_probe)(),
                 drv_info_t * drv_info, void * probe_id,
                 hw_path_t * hw_path, struct ics_table_type * isc,
                 int probe_type, char * name, char * desc);
\end{verbatim}

PARAMETERS

\texttt{this_node} \hspace{1em} A pointer to an \texttt{io\_tree\_node} struct.
\texttt{dev_probe} \hspace{1em} Probe function registered by device driver to be called by \texttt{driver\_addr\_probe()}. 
\texttt{drv_info} \hspace{1em} The \texttt{drv\_info\_t} struct registered with \texttt{wsio\_install()}. 
\texttt{probe_id} \hspace{1em} A unique identifier (for example, first 4 bytes of SCSI Inquiry data).
\texttt{hw_path} \hspace{1em} A pointer to a structure containing the hardware path information of the module being probed.
\texttt{isc} \hspace{1em} A pointer to the ISC structure assigned to the interface node that is being probed.
\texttt{probe_type} \hspace{1em} The type of hardware probe to perform.
\begin{itemize}
  \item \texttt{PROBE\_FIRST} \hspace{1em} Start at first available address.
  \item \texttt{PROBE\_NEXT} \hspace{1em} Increment the last address and start looking from there.
  \item \texttt{PROBE\_ADDRESS} \hspace{1em} Look only for this address.
\end{itemize}
\texttt{name} \hspace{1em} A string describing the device.
**desc**

A string describing the device found by the probe (usually 8 bytes of Vendor Id followed by 16 bytes of Product Id).

**DESCRIPTION**

The `driver_addr_probe()` WSIO function is provided by the driver writer. It can have any unique name. If you need this function, you register it with WSIO by executing the `wsio_register_addr_probe()` routine as part of your interface `driver_attach()` routine. Commonly, `driver` is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

**RETURN VALUES**

The `driver_addr_probe()` routine is expected to return the following values:

PROBE_SUCCESS Successfully found something identified it.

PROBE_UNSUCCESSFUL Got to end of the appropriate address range without finding anything, or something went wrong with the probe.

**CONSTRAINTS**

**WARNINGS**

The parameters shown in the call to `driver_addr_probe()` are only valid if this routine was previously registered with `wsio_register_addr_probe()`.

**SEE ALSO**

`driver_class_probe(WSIO_DRV)`, `wsio_probe_dev_info(WSIO4)`, `wsio_register_probe_func(WSIO3)`, `wsio_register_addr_probe(WSIO3)`
NAME

`driver_attach(WSIO_DRV)` - Claim a device for a driver.

SYNOPSIS

```c
int driver_attach (uint32_t card_id, struct isc_table_type *isc);
```

PARAMETERS

- `card_id` A four-byte card or product identifier.
- `isc` A pointer to the ISC structure assigned to the interface node that is being probed.

DESCRIPTION

The `driver_attach()` WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by including it in an attach chain with the driver’s `driver_install()` routine. Commonly, `driver` is replaced by your driver’s name.

The kernel searches the I/O backplane for hardware. When it finds a device, it first does preliminary initialization. Then it calls the `driver_attach()` routine at the head of the corresponding attach chain, e.g., `eisa_attach()`, that was created by the `driver_install()` routine.

Each `driver_attach()` routine in the chain looks at the `card_id`. If it recognizes the device as its own, it claims the device with the `isc_claim()` function, optionally puts a pointer to its `driver_if_init()` routine in `isc->gfsw->init`, and performs any other appropriate initialization. Then, whether it claims the device or not, it passes the same parameters to the next routine in the chain, using the routine name it saved in the `driver_install()` routine.

Since `isc_claim()` sets the INITIALIZED flag in `isc->if_info->flags`, you can also test this flag to see if there was a prior claim.


RETURN VALUES

Each `driver_attach()` routine is expected to return the return value.
returned by the next `driver_attach()` routine in the chain. The end-of-chain function returns a unique completion code.

**LP64 Considerations**

Be very careful with the `card_id` parameter. It is tempting to just define it as `PCI_ID` in a PCI `driver_attach()` routine. Due to 64-bit kernel parameter passing conventions you need to use code similar to the following example in determining that the device is your driver's:

```c
int zzz_attach(uint32_t idparm, struct isc_table_type *isc) {
    PCI_ID *id = (PCI_ID *)&idparm;
    if ((id->vendor_id != MY_VENDOR_ID&&
         (id->device_ID != MY_DEV_ID))
        return my_saved_attach(idparm, isc);
    else {
        /* code to claim card - set up isr, etc. */
        return my_saved_attach(idparm, isc);
    }
}
```

**CONSTRAINTS**

**SEE ALSO**

`get_new_isc(WSIO3), isc_table_type(KER4)`
NAME

`driver_class_probe` (WSIO_DRV) - Provide a driver-specific probing function.

SYNOPSIS

```c
#include <sys/wsio.h>
#include <sys/ioparams.h>
driver_class_probe (void * this_node, drv_info_t * drv_info, void * probe_id, hw_path_t * hw_path, struct ics_table_type * isc, int probe_type, char * dev_class, char * desc);
```

PARAMETERS

- **this_node** - A pointer to an `io_tree_node` struct.
- **drv_info** - The `drv_info_t` struct registered with `wsio_install()`.
- **probe_id** - A unique identifier (for example, first 4 bytes of SCSI Inquiry data).
- **hw_path** - A pointer to a structure containing the hardware path information of the module being probed.
- **isc** - A pointer to the ISC structure assigned to the interface node that is being probed.
- **probe_type** - The type of hardware probe to perform.
  - Defined types are:
    - `PROBE_FIRST` - Start at first available address.
    - `PROBE_NEXT` - Increment the last address and start looking from there.
    - `PROBE_ADDRESS` - Look only for this address.
- **dev_class** - A string describing the class of the device.
- **desc** - A string describing the device found by the probe (usually 8 bytes of Vendor Id followed by 16 bytes of Product Id).
DESCRIPTION

The `driver_class_probe()` WSIO function is provided by the driver writer. It can have any unique name. If you need one, you register it with WSIO by executing the `wsio_register_probe_func()` routine as part of your `driver_attach()` routine. Commonly, `driver` is replaced by your driver’s name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

The `driver_probe()` routine is expected to return the following values:

- **PROBE_SUCCESS**  Successfully found something and can identify it.
- **PROBE_UNSUCCESSFUL**  Got to end of the appropriate address range without finding anything, or something went wrong with the probe.

CONSTRAINTS

SEE ALSO

`driver_addr_probe(WSIO_DRV)`,
`wsio_register_addr_probe(WSIO4)`, `wsio_probe_dev_info(WSIO4)`,
`wsio_register_probe_func(WSIO3)`
NAME

driver_close(WSIO_DRV) – Close a device

SYNOPSIS

int driver_close(dev_t dev, int flag);

PARAMETERS

dev The device number of the file to be closed. The
driver_close() routine can extract the major and
minor numbers from the device number (see
major(WSIO3) and minor(WSIO3)).

(A user process specifies a file descriptor in the
close() system call.)

flag A value corresponding to the flag field in the
driver_open() call. See driver_open(WSIO_DRV).

DESCRIPTION

The driver_close() WSIO function is provided by the driver writer. It
can have any unique name. You pass the name to WSIO Services by
specifying it in the d_close field of the drv_ops structure. Commonly,
driver is replaced by your driver’s name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

The file system function which calls your driver through the bdevsw or
cdevsw tables always returns success (0) to the higher level file system
function which called it, ignoring the return value it gets from your
driver.

Therefore, the driver_close() routine need not return a valid value.
However, to avoid problems (as with strict compiler return value
checking), the driver_close() routine should return some integer
value.
CONRAINTS

SEE ALSO

close(2), driver_open(WSIO_DRV), drv_ops(CDIO4), open(2)
NAME

\texttt{driver\_dev\_init}(\texttt{WSIO\_DRV}) - Initialize a device driver

SYNOPSIS

\begin{verbatim}
text \texttt{driver\_dev\_init()};
\end{verbatim}

PARAMETERS

None.

DESCRIPTION

The \texttt{driver\_dev\_init()} WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by including it in the init chain, \texttt{dev\_init()}, with the \texttt{driver\_install()} routine. Commonly, \texttt{driver} is replaced by your driver's name. See HP-UX Driver Development Guide for details;

RETURN VALUES

Each \texttt{driver\_dev\_init()} routine is expected to return the return value returned by the next \texttt{driver\_dev\_init()} routine in the chain. The end-of-chain function returns a unique completion code.

CONSTRAINTS

SEE ALSO

\texttt{driver\_install}(\texttt{WSIO\_DRV})
NAME

driver_if_init (WSIO_DRV) – Initialize interface driver

SYNOPSIS

int driver_if_init (struct isc_table_type * isc);

PARAMETERS

isc

Pointer to an ISC structure for an interface this driver controls.

DESCRIPTION

The driver_if_init() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the isc->gfsw->init of the ISC structure with your driver's driver_attach() routine. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion.
-1 Error.

CONSTRAINTS

SEE ALSO

driver_attach (WSIO_DRV), isc_table_type (KER4),
isrlink (WSIO3)
NAME

driver_install(WSIO_DRV) – Register a driver with the system

SYNOPSIS

int driver_install();

PARAMETERS

None.

DESCRIPTION

The driver_install() WSIO function is provided by the driver writer. The name must be in the format shown, with driver replaced by the name of your driver as you specify it in the system file (defaults to /stand/system) and in the $DRIVER_INSTALL section of a master file in the /usr/conf/master.d directory.

See HP-UX Driver Development Guide for details;

RETURN VALUES

driver_install() is expected to return the value returned by wsio_install_driver() or install_driver().

Those values are:

0  Failure. The driver was not installed.
1  Success.

If it fails, the appropriate message below appears on the system console and in the system's error-log file. driver is the name of your driver.

wsio_install_driver: Install of driver driver failed.
install_driver: Install of driver driver failed.

CONSTRAINTS
SEE ALSO

config(1M), driver_attach(WSIO_DRV),
driver_dev_init(WSIO_DRV), driver_install(WSIO_DRV),
driver_probe(WSIO_DRV), install_driver(CDIO3), master(4),
wsio_install_driver(WSIO3), wsio_register_addr_probe(WSIO3),
wsio_register_probe_func(WSIO3)
NAME

\texttt{driver\_ioctl(WSIO\_DRV)} – Execute driver-specific control functions

SYNOPSIS

\begin{verbatim}
int driver_ioctl (dev_t dev, int cmd, caddr_t arg_ptr, int flag);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \texttt{dev} \hspace{1em} The device number of the associated device.
  \item \texttt{cmd} \hspace{1em} The command word described in DESCRIPTION.
  \item \texttt{arg\_ptr} \hspace{1em} Pointer to the command's arguments, if any.
  \item \texttt{flag} \hspace{1em} The file access flags. Most drivers ignore this parameter.
\end{itemize}

DESCRIPTION

The \texttt{driver\_ioctl()} WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the \texttt{d\_ioctl} field of the \texttt{drv\_ops} structure. Commonly, \texttt{driver} is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

\texttt{driver\_ioctl()} is expected to return the following values:

\begin{itemize}
  \item 0 \hspace{1em} Successful completion
  \item \textless 0 \hspace{1em} Error. The value is expected to be an error value.
\end{itemize}

CONSTRAINTS

LP64 Considerations

Pay particular attention to the \texttt{cmd} argument which has different values depending upon the calling program environment. HP-UX Driver
Development Guide

SEE ALSO

drv_ops(CDIO4), errno(2), ioctl(2), ioctl(5)
NAME

driver_isr(WSIO_DRV) – Execute device interrupt in interrupt context

SYNOPSIS

int driver_isr (isc_table_type isc, int arg1, int arg2);

PARAMETERS

isc A pointer to the isc_table_type structure passed in with your isrlink() call.
arg1 First optional parameter passed in with your isrlink() call.
arg2 Second optional parameter passed in with your isrlink() call.

DESCRIPTION

The driver_isr() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it as a parameter of the isrlink() function, executed in your driver_attach() or driver_if_init() routine. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

driver_isr() is expected to return the following values:

0 The card does not belong to this driver.
1 This routine handled the interrupt.

CONSTRAINTS

SEE ALSO

isrlink(WSIO3)
NAME

driver_minor_build(WSIO_DRV) - Build a minor number

SYNOPSIS

#include <sys/wsio.h>
#include <sys/ioparams.h>

int driver_minor_build (isc_table_type *isc, hw_path_t *dev_path, 
                    char *option);

PARAMETERS

isc A pointer to the ISC structure associated with the interface card for the device.

dev_path A pointer to a structure containing device hardware path information relative to the interface card.

option A null-terminated string indicating device-specific options.

DESCRIPTION

The driver_minor_build() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the drv_minor_build field of the wsio_drv_data_t structure. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

driver_minor_build() is expected to return the following values:

>0 Successful completion. The value is the minor number.

-1 Error. The minor number could not be built.
CONTRAINTS

SEE ALSO

wsio_drv_data(WSIO4)
NAME

driver_minphys (WSIO_DRV) – Driver specific transfer size adjustment

SYNOPSIS

void driver_minphys (struct buf *buf);

PARAMETERS

bp Transfer information structure;

DESCRIPTION

The driver_minphys() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to physio() by specifying it in the mincnt parameter in the call to physio(). Commonly, driver is replaced by your driver's name.

The driver_minphys() WSIO function adjusts the bp->b_bcount field of the buf structure passed in.

RETURN VALUES

driver_minphys() is a void function.

EXAMPLES

The following example illustrates a minphys() routine for a device that can handle at most NBPG size transfers.

void mydriver_minphys(struct buf *bp) {
    if (bp->b_bcount > NBPG)
        bp->b_bcount = NBPG;
}

CONSTRAINTS

SEE ALSO

minphys(KER2)
NAME

driver_open (WSIO_DRV) – Open a device

SYNOPSIS

int driver_open (dev_t dev, int oflags intptr_t dev, int mode);

PARAMETERS

dev The dev_t device number of the file to be opened. The
driver_open() routine can extract the major and
minor numbers from the device number. See
major (WSIO3) and minor (WSIO3).

oflags A value corresponding to the oflag parameter of the
open() system call. The kernel executes the oflag
functions (described in fcntl(5) and open(2)) before it
calls your driver. Your driver, therefore, can usually
ignore these flags.

Nevertheless, the kernel translates the O_xxxx values
into corresponding Fxxxx values, which it passes to the
driver_open() routine. The flags of possible interest
to your driver include: FREAD, FWRITE, FNDELAY, and
FEXCL.

mode Whether this is a call to a block or char driver. This
parameter is not accessible from an OPEN(2) call.

dummy A parameter used as by some drivers, though it is not
accessible from an OPEN(2) call

DESCRIPTION

The driver_open() WSIO function is provided by the driver writer. It
can have any unique name. You pass the name to WSIO Services by
specifying it in the driver_open field of the drv_ops structure.
Commonly, driver is replaced by your driver’s name.

See HP-UX Driver Development Guide for details;
RETURN VALUES

driver_open() is expected to return the following values:

0   Successful completion.
<>0   Error. The value is expected to be an errno value.

If the driver_open() routine is successful, the kernel’s open() call returns a file descriptor to the user. If it is unsuccessful, the kernel returns -1 to the user and sets errno to the value returned by the driver_open() routine. The user’s process can check the returned value and errno to determine whether an error occurred. See the <errno.h> header file for possible values for errno.

The driver_open() routine should return an error under these conditions. See open(2) for the expected error names.

• The device is off line.
• The device does not exist.
• The device was never configured into the system.
• The initialization of the device failed.
• The device is an exclusive-open device, and it is already open.

CONSTRAINTS

SEE ALSO

drv_ops(CDIO4), open(2)
NAME

driver_psize(WSIO_DRV) – Get swap partition size of a device

SYNOPSIS

int driver_psize(dev_t dev);

PARAMETERS

dev Contains encoded major and minor numbers;

DESCRIPTION

The driver_psize() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_psize field of the drv_ops structure. Commonly, driver is replaced by your driver’s name.

The driver_psize() WSIO function should return the size of the swap partition on a block swapping device, it is called by the kernel. Consider writing this routine only if your device is used for swapping.

See HP-UX Driver Development Guide for details;

RETURN VALUES

driver_psize() returns the following values:

>0 Successful completion. The value is the swap partition size.

-1 Error.

CONSTRAINTS

SEE ALSO

drv_ops(CDIO4)
NAME

\textit{driver\_read}(WSIO\_DRV) – Read data from/to a character device

SYNOPSIS

\texttt{int driver\_read (dev\_t dev, struct uio \* uio);} \\

PARAMETERS

\textit{dev} \hspace{1cm} The device number of the associated device file. The \texttt{routine can extract the major and minor numbers from the device number. Your \textit{driver\_open()} routine should verify that the minor number is valid.}

\textit{uio} \hspace{1cm} A pointer to a \texttt{uio} structure. The \texttt{uio} structure contains information about the data being read or written.

DESCRIPTION

When a user process issues a \texttt{read()}, \texttt{readv()}, \texttt{write()}, or \texttt{writev()} system call for a character device, the kernel puts information about the request in the \texttt{uio} and \texttt{iovec} structures and dispatches control to the \textit{driver\_read()} or \textit{driver\_write()} routine for that device, passing the \texttt{uio} structure to the driver as a parameter.

See HP-UX Driver Development Guide for details;

RETURN VALUES

\textit{driver\_read()} and \textit{driver\_write()} are executed to return the following values:

\begin{itemize}
  \item \texttt{0} \hspace{1cm} Successful completion.
  \item \texttt{<>0} \hspace{1cm} Error. The value is expected to be an \texttt{errno} value.
\end{itemize}

CONSTRAINTS
EXAMPLES

See physio(KER2) and uiomove(KER2).

SEE ALSO

drv_ops(CDIO4), physio(KER2), uiomove(KER2)
NAME

driver_select(WSIO_DRV) - Test I/O completion on a device

SYNOPSIS

int driver_select (dev_t dev, int flag);

PARAMETERS

dev The device number.
flag The type of readiness to test, according to the following values:
    FREAD Read
    FWRITE Write
    0 Exception conditions

DESCRIPTION

The driver_select() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_select field of the drv_ops structure. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

driver_select() is expected to return the following values:

<>0 True. The device or driver is ready for read or write or an exception condition was found. The kernel sets the corresponding bit in the bit-mask field that select() returns to the user.

0 False. The device or driver is not ready for read or write or no exception condition was found. select() puts the calling process to sleep until the condition becomes true. The driver must inform the system when this condition becomes true.

If the driver_select() routine detects an error while selecting for read
or write, it should return false and set an error in \texttt{u.u\_error}. If it detects an error while selecting for an exception condition, it should return true and set an error in \texttt{u.u\_error}.

\section*{Constraints}

\section*{See Also}

drv\_ops(CDIO4), selwakeup(KER2), select(2)
NAME

driver_strategy (WSIO_DRV – Execute block read or write for character or block devices

SYNOPSIS

void driver_strategy (struct buf * bp);

PARAMETERS

bp A pointer to a buf structure, which contains all the information that the driver_strategy() routine needs to process the request.

DESCRIPTION

The driver_strategy() WSIO function is provided by the driver writer. It can have any unique name. For a block device, you pass the name to WSIO Services by specifying it in the driver_strategy field of the drv_ops structure. For a character device, you pass the name as a parameter to physio(). Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

physio(KER2)
NAME

\texttt{driver\_write(WSIO\_DRV)} – Write data from/to a character device

SYNOPSIS

\begin{verbatim}
int driver_write (dev_t dev, struct uio * uio);
\end{verbatim}

PARAMETERS

- \textit{dev} The device number of the associated device file. The routine can extract the major and minor numbers from the device number. Your \textit{driver\_open()} routine should verify that the minor number is valid.
- \textit{uio} A pointer to a \textit{uio} structure. The \textit{uio} structure contains information about the data being read or written.

DESCRIPTION

When a user process issues a \textit{read()}, \textit{readv()}, \textit{write()}, or \textit{writev()} system call for a character device, the kernel puts information about the request in the \textit{uio} and \textit{iovec} structures and dispatches control to the \textit{driver\_read()} or \textit{driver\_write()} routine for that device, passing the \textit{uio} structure to the driver as a parameter.

See HP-UX Driver Development Guide for details;

RETURN VALUES

\textit{driver\_read()} and \textit{driver\_write()} are executed to return the following values:

\begin{verbatim}
0      Successful completion.
<>0    Error. The value is expected to be an \textit{errno} value.
\end{verbatim}

CONSTRAINTS
EXAMPLES

See physio(KER2) and uiomove(KER2).

SEE ALSO

drv_ops(CDIO4), physio(KER2), uiomove(KER2)
NAME

free_isc(WSIO3) - Free a driver's ISC entry

SYNOPSIS

#include <sys/wsio.h>

int free_isc (struct isc_table_type *isc);

PARAMETERS

isc Pointer to an ISC entry.

DESCRIPTION

The free_isc() kernel function frees an ISC entry that was obtained explicitly by a driver by using get_new_isc() or another similar service. free_isc() should be called after a severe driver-disabling error or before a driver is unloaded.

RETURN VALUES

free_isc() returns the following values:

0 Successful completion.
-1 Error.

CONSTRAINTS

SEE ALSO

get_new_isc(WSIO3)
NAME

get_new_isc (WSIO3) - Allocate a new ISC structure for this card function

SYNOPSIS

#include <sys/io.h>

struct isc_table_type * get_new_isc (struct isc_table_type * dd_isc);

PARAMETERS

dd_isc Pointer to a currently allocated ISC structure.

RETURN VALUES

CONSTRAINTS

DESCRIPTION

The get_new_isc() WSIO function allocates a new ISC structure when you need more than one, as for a multifunction card.

If the isc->ftn_no field is not -1 in an entry for a multifunction card, the driver_attach() routine should call get_new_isc() to allocate a new ISC structure for the driver’s functions and set the isc->ftn_no field of the new ISC structure to the function number for its portion of the card, and then continue its normal power-on initializations, using the new ISC structure that was returned from get_new_isc().

Pass the new ISC on to the next driver in the attach chain.

The get_new_isc() function allocates and zeros out a new ISC structure and then does the following:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus_type</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>my_isc</td>
<td>Copied from old_isc</td>
</tr>
</tbody>
</table>
if_reg_ptr Copied from old_isc
bus_info Copied from old_isc
ftn_no Set to -1, the caller should correctly set this field after call
old_isc->next_ftn Set to the new isc
if_info Allocated and then copied from old_isc
new->next_ftn Set to NULL
ifsw Copied from old_isc
if_drv_data Copied from old_isc
gfsw Allocated and copied from old_isc if old_isc->gfsw is not NULL

RETURN VALUES

get_new_isc() returns the following values:
<>NULL Success. The value is a pointer to a new ISC structure.
NULL Failure. get_new_isc() was unable to allocate memory for the new ISC structure.
NAME

iodone(WSIO3) - Complete the buffer I/O transaction

SYNOPSIS

#include <sys/buf.h>

int iodone (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The iodone() WSIO function is used by legacy drivers as an alias for biodone(). New drivers should call biodone() directly instead of calling iodone().

The biodone() kernel function completes the buffer I/O transaction. There should be a corresponding call to biowait() for the same bp.

If B_CALL is set in bp->b_flags, biodone() calls the callback function specified in bp->b iodone. The callback function is expected to set the B_DONE flag in bp->b_flags.

If B_CALL is not set in bp->b_flags, biodone() marks the buffer I/O as completed by setting the B_DONE flag in bp->b_flags. If B_ASYNC is set, biodone() releases the buf structure and associated buffer pointed to by bp, else it resumes the thread waiting on the corresponding call to biowait().

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock of order \textgreater;= \text{BUF\_HASH\_LOCK\_ORDER}.
WARNINGS

biodone() calls panic() if B_DONE is set in bp->b_flags upon entry.

SEE ALSO

biodone(KER2), biowait(KER2), iowait(WSI03)
NAME

iowait(WSIO3) – Wait for the buffer I/O to complete.

SYNOPSIS

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

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

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The iowait() WSIO function is used by legacy drivers as an alias for biowait(). New drivers should call biowait() directly instead of calling iowait().

The biowait() kernel function waits for the completion of the buffer I/O specified by bp. A corresponding call to biodone() is required to resume the waiting thread.

RETURN VALUES

iowait() returns the following values:

- 0 Must not be called in an interrupt context.
- <>0 Error.

CONSTRAINTS

Must not be called in an interrupt context.

Must not be called while holding a spinlock.

EXAMPLES

```
int error;
struct buf *bp;

error = iowait(bp);
```
/*  
* iowait() returns 0 if the IO completes successfully.
* A non-zero value is returned if an error has been 
* encountered, however, the error value returned is not
* always for the IO completion. To get the IO 
* completion error that is returned with the buf, we 
* need to call geterror().
*/
if (error) {
    error = geterror(bp);
}

SEE ALSO

biodone(KER2), biowait(KER2), geterror(KER2), iodone(WSIO3)
NAME
isc_claim(WSIO3) – Marks an ISC entry as claimed by the driver.

SYNOPSIS
#include <sys/io.h>
#include <sys/wsio.h>

void isc_claim (struct isc_table_type *isc, wsio_drv_info_t drv_info);

PARAMETERS
isc Pointer to the ISC entry associated with an interface card or device.
drv_info Pointer to the wsio_drv_info_t structure of the driver that is claiming the ISC structure.

DESCRIPTION
The isc_claim() WSIO function marks an ISC entry as claimed by the driver. isc_claim() is called in the driver_attach function when the driver wants to be assigned to the device represented by the ISC entry.

If drv_info is NULL, the driver is indicating the ISC entry should be discarded. An example of this situation is when the PS2 keyboard driver encounters its second ISC entry. Since the driver only uses the first ISC entry, it can claim and discard the second ISC entry by passing NULL as drv_info.

RETURN VALUES
None.

CONSTRAINTS

EXAMPLE
static int
mydrv_attach(uint32_t id, struct isc_table_type *isc)
{
    ...
    if (id == MY_DEVICE_HW_ID) {
        /*
         * Specify the interface init function that is
         * called for each claimed ISC entry after the
         * attach chain processing has completed.
         */
        isc->gfsw->init = mydrv_if_init;
        /*
         * Claim the ISC entry representing the device.
         */
        isc_claim(isc, &mydrv_info);
    }
    /*
    * Call the next driver on the attach chain.
    */
    return (*mydrv_saved_attach)(id, isc);
}

SEE ALSO

driver_attach(WSIO_DRV)
NAME

isrlink(WSIO3) - Register an interrupt service routine

SYNOPSIS

```c
int isrlink (struct isc_table_type *isc, int (*isr)(),
            int irq_line, long arg1, long arg2);
```

PARAMETERS

- `isc`  Pointer to the ISC structure for the driver.
- `isr`  Pointer to the driver's interrupt service routine.
- `irq_line`  Interrupt request line asserted by the device. For PCI devices this should be -1 allowing WSIO services to determine the interrupt request line being used.
- `arg1`  Driver defined parameter passed as the first parameter to `isr`. Typically, `isc` is passed as `arg1`.
- `arg2`  Driver defined parameter passed as the second parameter to `isr`.

DESCRIPTION

The `isrlink()` WSIO function registers an interrupt service routine (ISR). `isrlink()` is typically called in the `driver_if_init()` function, if specified by `driver_attach()` or in `driver_attach()`.

When `isrlink()` returns, interrupts for the assigned `irq_line` are enabled. The driver should be prepared to handle an interrupt from its device or another device sharing the `irq_line`.

RETURN VALUES

- `isrlink()` returns the following values:
  - `0`  Successful completion.
  - `WSIO_ERROR`  Error.
CONSTRAINTS

EXAMPLE

```c
static int
mydrv_if_init(struct isc_table_type *isc)
{
    ...
    mydrv_reset_hw(isc);
    return isrlink(isc, mydrv_isr, -1, (long)isc, 0L);
}
```

SEE ALSO

`driver_attach(WSIO_DRV), driver_if_init(WSIO_DRV),
driver_isr(WSIO_DRV), isrunlink(WSIO3)`.
NAME

isrunlink(WSIO3) - Remove the ISR registered by isrlink()

SYNOPSIS

```c
#include<sys/wsio.h>

int isrunlink (struct isc_table_type *isc, int (*isr)(),
               int irq_line, long arg1, long arg2);
```

PARAMETERS

- `isc` : Pointer to the ISC structure for the driver.
- `isr` : Pointer to the driver's interrupt service routine.
- `irq_line` : Interrupt request line asserted by the device. For PCI devices this should be -1, allowing WSIO services to determine the interrupt request line used.
- `arg1` : Driver defined parameter passed as the first parameter to ISR.
- `arg2` : Driver defined parameter passed as the second parameter to ISR.

DESCRIPTION

The `isrunlink()` WSIO function removes the `isr` registered by `isrlink()`. This function should be called before a driver is unloaded but after the device has been quiesced.

RETURN VALUES

isrunlink() returns the following values:

- `0` : Successful completion.
- `WSIO_ERROR` : Error.

CONSTRAINTS
SEE ALSO

isrlink(WSIO3)
NAME
   m_instance(WSIO3) – Get the driver instance field from the device number

SYNOPSIS
   #include <sys/io.h>
   #define m_instance(dev) (int)((unsigned) (dev)>>16&0xff)

PARAMETERS
   dev                The device number of a device.

DESCRIPTION
   The m_instance() WSIO macro returns the driver instance field from the device number.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

m_wsio_funcnum(WSI03) - Get the number of an interface card function

SYNOPSIS

int m_wsio_funcnum (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

- **dev**: The dev_t number of a device.
- **drv_hdr_ptr**: A pointer to the wsio_drv_info_t structure for the device.

DESCRIPTION

The m_wsio_funcnum() WSIO function returns the number of the interface card function associated with device number `dev`.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

m_wsio_selcode(WSI03) - Get the select code for a device

SYNOPSIS

int m_wsio_selcode (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

dev  The dev_t number of a device.

drv_hdr_ptr  A pointer to the wsio_drv_info_t structure for the device.

DESCRIPTION

The m_wsio_selcode() WSIO function returns the select code associated with device number dev.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

m_wsio_vsc(WSIO3) – Return the system bus module number for a device number

SYNOPSIS

int m_wsio_vsc (dev_t dev, wsio_drv_info_t * drv_hdr_ptr);

PARAMETERS

dev The dev_t number of a device.
drv_hdr_ptr A pointer to the wsio_drv_info_t structure for the device.

DESCRIPTION

The m_wsio_vsc() WSIO function returns the system bus module number for device number dev.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

mod_wsio_attach_list_add(WSIO3) - Add the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

int mod_wsio_attach_list_add (int type, void *attach_func);

PARAMETERS

type Type of WSIO attach list.
attach_func Pointer to the driver attach function.

DESCRIPTION

The mod_wsio_attach_list_add() WSIO function adds the driver attach function pointer attach_func to the WSIO attach list specified by type.

Dynamically loadable drivers call mod_wsio_attach_list_add() when they are loaded into the kernel.

The type parameter specifies the attach list to use. Valid values are:

- MOD_WSIO_CORE for Core I/O attach list.
- MOD_WSIO_EISA for EISA I/O attach list.
- MOD_WSIO_PCI for PCI I/O attach list.

The attach_func parameter points to the driver attach function that will be called by the WSIO environment to see if the driver wants to claim a device.

RETURN VALUES

mod_wsio_attach_list_add() returns the following values:

0 Successful completion.
1 Error.
CONSTRAINTS

EXAMPLE

/*
 * Add my driver attach function to the WSIO attach list
 * for claiming PCI devices.
 */
if (mod_wsio_attach_list_add(MOD_WSIO_PCI,
    &mydrv_attach)) {
    return ENXIO;    /* attach add failed! */
}

SEE ALSO

mod_wsio_attach_list_remove(WSIO3).
NAME

\texttt{mod\_wsio\_attach\_list\_remove} (WSIO3) – Remove the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

\begin{verbatim}
int mod_wsio_attach_list_remove (int type, void *attach_func);
\end{verbatim}

PARAMETERS

- \textit{type} Type of WSIO attach list.
- \textit{attach_func} Pointer to the driver attach function.

DESCRIPTION

The \texttt{mod\_wsio\_attach\_list\_remove()} WSIO function removes the driver attach function pointer \textit{attach_func} from the WSIO attach list specified by \textit{type}. The \textit{attach_func} and \textit{type} parameters must match the parameters passed to \texttt{mod\_wsio\_attach\_list\_add()}.

Dynamically loadable drivers call \texttt{mod\_wsio\_attach\_list\_remove()} when they are unloaded from the kernel.

The \textit{type} parameter specifies the attach list to use. Valid values are:

- \texttt{MOD\_WSIO\_CORE} for Core I/O attach list.
- \texttt{MOD\_WSIO\_EISA} for EISA I/O attach list.
- \texttt{MOD\_WSIO\_PCI} for PCI I/O attach list.

The \textit{attach_func} parameter points to the driver attach function.

RETURN VALUES

\texttt{mod\_wsio\_attach\_list\_remove()} returns the following values:

- \texttt{0} Successful completion.
- \texttt{1} Error.

CONSTRAINTS
EXAMPLE

/*
 * Remove my driver attach function from the WSIO attach list
 * for claiming PCI devices.
 */
if (mod_wsio_attach_list_remove(MOD_WSIO_PCI,
       &mydrv_attach)) {
    return ENXIO;  /* attach remove failed! */
}

SEE ALSO

mod_wsio_attach_list_add(WSIO3).
NAME

wsio_activate_probe(WSIO3) – Activate the probe function for a driver.

SYNOPSIS

void wsio_activate_probe (char * probe_name,
                        struct drv_info* drv_infop);

PARAMETERS

probe_name Name of the device probe function as registered by
            wsio_register_dev_probe()

drv_infop Pointer to the driver drv_info structure

DESCRIPTION

The wsio_activate_probe() WSIO function connects the probe
function for a dynamically loadable interface driver to the driver
drv_info structure wsio_activate_probe() is called in the driver’s
load entry point after its device probe function has been registered with
the WSIO CDIO.

RETURN VALUES

None

CONSTRAINTS

EXAMPLES

static wsio_drv_info_t mydrv_info = { ... };

int mydrv_load(void * arg)
{
    /*
    * Use the drv_info passed to the driver as arg
    * instead of using the static version.
    */
}
mydrv_info.drv_info = (drv_info_t *)arg;

/*
 * Register the driver with WSIO.
 * Note: returns 0 on failure.
 */
if (!wsio_install_driver(&mydrv_info)) {
    return ENXIO; /* Install driver failed! */
}

/* Add my driver attach function to the WSIO attach list
 * for claiming PCI devices.
 */
if (mod_wsio_attach_list_add(MOD_WSIO_PCI, &mydrv_attach)) {
    /*
     * Attach list add failed! Uninstall the driver
     * and return.
     */
    (void)wsio_uninstall_driver(&mydrv_info);
    return ENXIO;
}

/* Register the device probe function for the driver.
 */
if (wsio_register_dev_probe(IF_CLASS, mydrv_probe_func, "mydrv_probe")) {
    /*
     * Register device probe failed! Remove driver from
     * the attach list, uninstall the driver and return.
     */
    (void)mod_wsio_attach_list_remove(MOD_WSIO_PCI, &mydrv_attach);
    (void)wsio_uninstall_driver(&mydrv_info);
    return ENXIO;
}

/*
 * The following step is only required for dynamically
 * loadable drivers: connect the probe function.
 */
wsio_activate_probe("mydrv_probe", mydrv_info.drv_info);
return 0;
SEE ALSO

wsio_register_dev_probe(WSIO3),
wsio_unregister_dev_probe(WSIO3)
NAME

wsio_allocate_shared_memory(WSIO4) – Allocate and map contiguous memory used for continuous DMA.

SYNOPSIS

#include <sys/wsio.h>

shmem_status_t wsio_allocate_shared_memory (  
struct isc_table_type *isc, size_t size,  
caddr_t *iova, caddr_t *vaddr,  
wsio_shmem_attr_t type);  

PARAMETERS

isc  Pointer to an ISC table entry.
size  Size in bytes or memory to be allocated.
iova  Pointer to the returned I/O virtual address.
vaddr  Pointer to the returned virtual address.
type  Bit mask of the requested memory attributes.

DESCRIPTION

The wsio_allocate_shared_memory() function allocates and maps contiguous memory used for continuous DMA. For packet DMA (short lived DMA typical of I/O transactions), temporary mappings should be done using wsio_map(). Continuous DMA is intended for accesses by a device on a continuous basis, typically for device control and status.

Platforms that implement I/O virtual addressing will allocate memory contiguous in I/O virtual address space; platforms that do not will allocate memory contiguous in physical address space. Some platforms may allocate memory that is local to an I/O adapter (aka a “bus bridge”), and such memory may be severely limited in size.

The isc parameter is a pointer to the ISC table entry assigned to the driver’s interface card. It is the handle for the driver instance.

The size parameter is the size in bytes of memory to be allocated.
The `iova` parameter is a pointer to the returned I/O virtual address. It is the base address of the allocated memory from the view of the interface card.

The `vaddr` parameter is a pointer to the returned virtual address corresponding to the I/O virtual address. It is the base address of the allocated memory from the view of the processor.

The `type` parameter is a bit mask of the requested memory attributes. Valid memory attributes are the following:

- `WSIO_SHMEM_OPTIMIZEDEVICE_LATENCY` - allocation should optimize for device access latency. If the platform allows, allocated memory should be local to the I/O adapter connecting the interface card. This is the default attribute if none are specified.
- `WSIO_SHMEM_OPTIMIZEHOST_LATENCY` - allocation should optimize for host access latency. If the platform allows, allocated memory should be in host memory.
- `WSIO_SHMEM_INBOUND` - hint indicating the allocated memory will be used exclusively for inbound (device to memory) DMA only
- `WSIO_SHMEM_OUTBOUND` - hint indicating the allocated memory will be used exclusively for outbound (memory to device) DMA only
- `WSIO_SHMEM_DEV_WEAK_OK` - hint indicating accesses to the allocated memory can be weakly ordered.
- `WSIO_SHMEM_ALIGN_ON_SIZE` - allocation must align the memory on the size specified. If, for example, size is a power of 2, the base address of the memory allocated must be aligned to the same power of 2 or a multiple of that value.

**RETURN VALUES**

`wsio_allocate_shared_memory()` returns the following values:

- `SHMEM_OK`  Successful completion
- `SHMEM_NO_RESOURCES`  Memory not allocated

**CONSTRAINTS**
EXAMPLES

caddr_t my_iova;
caddr_t my_vaddr;

/*
 * Allocate contiguous memory that is page bytes in size
 * and aligned to a page size boundary.  If the platform
 * allows, allocate memory that is local to the I/O
 * adapter (bus bridge) connecting the interface card.
 */
if (wsio_allocate_shared_memory(
    isc, NBPG, &my_iova, &my_vaddr,
    WSIO_SHMEM_OPTIMIZE_DEVICE_LATENCY |
    WSIO_SHMEM_ALIGN_ON_SIZE) != SHMEM_OK) {
    /*
     * Failed to allocate memory for continuous DMA.
     */
}

SEE ALSO

wsio_flush_shared_memory(WSIO4),
wsio_free_shared_memory(WSIO4)
NAME

\texttt{wsio_drv_data_t(WSIO4)} - Driver-specific fields for WSIO drivers

SYNOPSIS

DESCRIPTION

The \texttt{wsio_drv_data_t} WSIO structure type, defined in \texttt{<sys/wsio.h>}, contains driver-specific fields for WSIO drivers.

STRUCTURE MEMBERS

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{drv_path}</td>
<td>Follow these guidelines:</td>
</tr>
<tr>
<td></td>
<td>• For device drivers, \texttt{drv_path} is typically a string that contain the interface card's type and the device's class. For example, \texttt{scsi_disk}.</td>
</tr>
<tr>
<td></td>
<td>• For interface drivers, \texttt{drv_path} should match the card's type. For example, \texttt{scsi}.</td>
</tr>
<tr>
<td></td>
<td>• For pseudo drivers, \texttt{drv_path} should match the card's class. For example, \texttt{graphics}.</td>
</tr>
<tr>
<td>\texttt{drv_type}</td>
<td>One of the following values:</td>
</tr>
<tr>
<td></td>
<td>\texttt{T_INTERFACE} The driver controls an interface card.</td>
</tr>
<tr>
<td></td>
<td>\texttt{T_DEVICE} The driver controls a hardware device.</td>
</tr>
<tr>
<td>\texttt{drv_flags}</td>
<td>One of the following values:</td>
</tr>
<tr>
<td></td>
<td>\texttt{DRV_CONVERGED} The driver meets the HP-UX Release 10.0 Converged I/O specifications. All new drivers should meet these specifications.</td>
</tr>
<tr>
<td></td>
<td>\texttt{NOT_CONVERGED} The driver conforms to the pre-Release 10.0 unconverged specifications.</td>
</tr>
</tbody>
</table>
**Functions, Macros and Structures**

**drv_minor_build**

- Pointer to your minor number formatter. Use NULL if you don't provide one.

**drv_minor_decode**

- Pointer to your minor number interpreter. Use NULL if you don't provide one.

**EXAMPLES**

```c
static wsio_drv_data_t sdisk_data = {
    "scsi_disk",
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};
```

**RETURN VALUES**

**CONSTRAINTS**

**SEE ALSO**
NAME

wsio_drv_info(WSIO4) - Structure containing pointers to other CDIO and WSIO data structures

SYNOPSIS

#include <sys/wsio.h>

typedef struct wsio_drv_info {
    drv_info_t * drv_info;
    drv_ops_t * drv_ops;  // wsio_drv_data_t * drv_data;
    wsio_drv_info_t;
}

DESCRIPTION

The wsio_drv_info_t WSIO structure type, defined in <sys/wsio.h>, contains pointers to three other data structures.

STRUCTURE MEMBERS

- **drv_info**  Pointer to a drv_info_t CDIO structure.
- **drv_ops**   Pointer to a drv_ops_t CDIO structure.
- **drv_data**  Pointer to a wsio_drv_data_t structure.

RETURN VALUES

CONSTRAINTS

SEE ALSO

drv_info(CDIO4), drv_ops(CDIO4), wsio_drv_data_t(WSIO4)
NAME

wsio_fastmap(WSIO3) - Map all or part of a host address range into an I/O virtual address range

SYNOPSIS

#include <sys/dma.h>

int wsio_fastmap (struct isc_table_type *isc, int range_type,
                    struct iovec *host_range,
                    struct iovec *io_range);

PARAMETERS

isc               Pointer to an ISC structure.

range_type        The type of host address for host_range. If range_type > 0 then this is the space ID of the address range. Other values are:

                  KERNELSPACE        Kernel virtual buffer (same as space ID = 0)

host_range        A pointer/length pair indicating the host address range of type range_type. The length will be modified by the service to indicate the bytes remaining to be mapped. This length is also returned by wsio_fastmap().

io_range          A pointer/length pair filled by wsio_fastmap() with the I/O virtual address range or EISA address range.

DESCRIPTION

The wsio_fastmap() WSIO function provides the same functionality as wsio_map() when the entire host address range resides on a single physical page. This condition is not checked by wsio_fastmap(). Cache-line fragments are ignored (as in wsio_map() with the IO_IGN_ALIGNMENT hint).

Since the behavior and side effects are different for coherent and noncoherent systems, these are discussed separately below. Although the behaviors are slightly different, drivers should not have dependencies on
these differences. In all cases, the programming model is the same.

All mappings remain valid until the io_range is unmapped by calls to wsio_unmap(). If the io_range is reused, dma_sync() must be used to resynchronize it.

Behavior on Noncoherent Systems

On noncoherent systems, the I/O virtual address is equivalent to the host physical address. wsio_map() will return an io_range at each page break in the host_range. If the mapping is for an EISA module, then the io_range is filled with EISA addresses and the EISA map will contain the host physical address.

Behavior on Coherent Systems

On coherent systems, the I/O virtual address is obtained by creating a mapping in the I/O PDIR. For cache-line fragments, the SAFE bit will be set in the I/O PDIR, unless inhibited by IO_IGN_ALIGNMENT. However, it is expected that buflets will be used by the caller to handle them. If the mapping is for an EISA module, then the io_range is filled with EISA addresses and the EISA map will contain the corresponding IOVAs.

RETURN VALUES

wsio_fastmap() always returns 0 on a Noncoherent system.

wsio_fastmap() returns the following values on a Coherent system:

0       The range was fully mapped.
-1      The necessary resources could not be obtained.

CONSTRAINTS

WARNING

The IO_NO_SEQ flag is NOT set with wsio_fastmap(), and if the mapping is cache line aligned the IO_SAFE bit is NOT set. Under certain conditions this call MUST NOT be used for PCI. See pci-errata(PCI5) for details.

EXAMPLE

The following function maps a single page of memory (virt_page is aligned on a page boundary):
caddr_t my_page_map(port_num, caddr_t virt_page)
    port_num_type port_num;
    caddr_t virt_page;
    {
        struct iovec host, io;
        host->iov_base = virt_page;
        /* virt_page is page-aligned */
        host->iov_len = NBPG;

        wsio_fastmap(port_num, KERNELSPACE, &host, &io);
        return io->iov_base;
    }

WARNINGS

It is up to the caller to ensure that the space to be mapped does not cross a page boundary. On a Coherent system this will be indicated by a return of -1, on a Noncoherent system the call will succeed.

SEE ALSO

dma_sync(CDIO3), init_map_context(CDIO3),
isc_table_type(KER4), pci_errata(PCI5), wsio_fastmap(WSIO3),
wsio_remap(WSIO3), wsio_set_attributes(WSIO3),
wsio_unmap(WSIO3)
NAME

wsio_flush_shared_memory(WSIO) - Flush the memory previously allocated and mapped by \texttt{wsio\_allocate\_shared\_memory()}. 

SYNOPSIS

\begin{verbatim}
#include <sys/wsio.h>
shmem_status_t wsio_flush_shared_memory (
    struct isc_table_type *isc, size_t size, 
    caddr_t iova, caddr_t vaddr, 
    wsio_shmem_attr_t type);
\end{verbatim}

PARAMETERS

- \texttt{isc} \hspace{1cm} Pointer to an ISC table entry.
- \texttt{size} \hspace{1cm} Size in bytes or memory to be flushed.
- \texttt{iova} \hspace{1cm} I/O virtual address.
- \texttt{vaddr} \hspace{1cm} Virtual address corresponding to \texttt{iova}.
- \texttt{type} \hspace{1cm} Bit mask of the memory attributes.

DESCRIPTION

The \texttt{wsio\_flush\_shared\_memory()} function flushes the memory previously allocated and mapped by \texttt{wsio\_allocate\_shared\_memory()}. This ensures that data in the allocated memory is viewed consistently by the device and processors. All parameters passed to \texttt{wsio\_flush\_shared\_memory()} must match the parameters passed to the corresponding call to \texttt{wsio\_allocate\_shared\_memory()}. 

RETURN VALUES

\texttt{wsio\_flush\_shared\_memory()} returns the following values:

- \texttt{SHMEM\_OK} \hspace{1cm} Successful completion
- \texttt{SHMEM\_NO\_RESOURCES} \hspace{1cm} Memory not flushed
CONSTRAINTS

SEE ALSO

wsio_allocate_shared_memory (WSIO4),
wsio_free_shared_memory (WSIO4)
NAME

wsio_free_shared_memory(WSIO4) - Release and unmap contiguous memory previously allocated and mapped by wsio_allocate_shared_memory().

SYNOPSIS

```c
#include <sys/wsio.h>

void wsio_free_shared_memory (struct isc_table_type *isc,
                             size_t size, caddr_t iova, caddr_t vaddr,
                             wsio_shmem_attr_t type);
```

PARAMETERS

- **isc**: Pointer to an ISC table entry.
- **size**: Size in bytes or memory to be released.
- **iova**: I/O virtual address.
- **vaddr**: Virtual address corresponding to iova.
- **type**: Bit mask of the memory attributes.

DESCRIPTION

The `wsio_free_shared_memory()` function releases and unmaps contiguous memory previously allocated and mapped by `wsio_allocate_shared_memory()`. All parameters passed to `wsio_free_shared_memory()` must match the parameters passed to the corresponding call to `wsio_allocate_shared_memory()`.

RETURN VALUES

None

CONSTRAINTS

None
SEE ALSO

wsio_allocate_shared_memory(WSIO4),
wsio_flush_shared_memory(WSIO4)
NAME

wsio_get_interrupts(WSIO3) - Determine which interrupt has been assigned to a card

SYNOPSIS

#include <sys/io.h>

int wsio_get_interrupts (struct isc_table_type *isc);

PARAMETERS

isc A pointer to the ISC structure associated with the interface card.

DESCRIPTION

The wsio_get_interrupts() WSIO function determines the IRQ of an interface card hardware module associated with the given ISC structure. It expects the "interrupt" property of the iotree node to have been appropriately assigned during I/O configuration. It can be used by drivers for getting the information needed to set up their isc->eim values and for setting up their isrlink() service calls. PCI bus drivers must use -1 for their isrlink() calls, allowing the PCI services to determine the actual interrupt line assigned.

RETURN VALUES

wsio_get_interrupts() returns the following values:

>0 The IRQ for the card.

-1 isc is NULL or there was a problem retrieving the node's interrupt property.

CONSTRAINTS

EXAMPLES

mydriver_init(isc)
struct isc_table_type *isc;
{
    int wsio_eim;
    ...
    if (NOT_PCI) {
        mydriver_reset(isc);
        isc->eim = wsio_eim = wsio_get_interrupts(isc);
        isc->eim_control = 0;
    } else {
        mydriver_reset(isc);
        wsio_eim = -1;
    }
    isrlink(isc,mydriver_isr,wsio_eim,isc,isc->if_drv_data);
    ...
}

SEE ALSO

isc_table_type(KER4)
NAME

wsio_get_isc (WSIO3) – Retrieve the ISC structure pointer for a device file.

SYNOPSIS

```c
#include <sys/io.h>
#include <sys/wsio.h>

int wsio_get_isc (dev_t dev, struct isc_table_type **isc_ptr, 
                 wsio_drv_info_t *wsio_drv_info);
```

PARAMETERS

- `dev` The device file of the hardware for which we want an ISC pointer.
- `isc_ptr` A pointer to the location for the routine to put a pointer to the ISC structure.
- `wsio_drv_info` A pointer to the `wsio_drv_info_t` header structure of the driver, used to decode `dev`. If a NULL value is passed in this field, `wsio_get_isc()` will use the `wsio_drv_info_t` structure of the character device (not block) whose major number matches that of the `dev` argument.

DESCRIPTION

The `wsio_get_isc()` WSIO function finds an ISC table entry associated with the nearest interface ancestor of the device specified by `dev`, assuming that `dev` uses the HP-UX Converged I/O minor number format.

NOTE

Drivers modified to match the Converged I/O driver guidelines will automatically have the old `get_isc()` call mapped to this one when they include the `wsio.h` header file. Drivers that have not been modified for Release 10.0 and do not include `wsio.h` will be assumed to use the old minor number format and the old `get_isc()`.

Modified drivers that still call the old `get_isc()` will work only for drivers that pass their character `dev` values. If a driver has only its block
dev, it must call wsio_get_isc() directly.

RETURN VALUES

1  Successful completion. The ISC pointer found is returned in isc_ptr.
0  Failure. The ISC could not be found.

CONSTRAINTS

SEE ALSO
NAME

wsio_get_pva(WSIO3) – Translate an I/O virtual address to its Processor virtual address

SYNOPSIS

#include <sys/dma.h>

caddr_t wsio_get_pva (struct isc_table_type *isc, caddr_t iova);

PARAMETERS

isc Pointer to an ISC structure.

iova An I/O Virtual Address.

DESCRIPTION

The wsio_get_pva() WSIO function translates I/O virtual addresses into processor virtual addresses. The translation is more efficient if the memory object is equivalently mapped (ProcVA == Physical address). This function is restricted to virtual buffers in kernel space.

RETURN VALUES

<>NULL The processor virtual address, if the I/O virtual address exists and there is a corresponding processor virtual address in KERNELSPACE.

NULL Otherwise.

CONSTRAINTS

EXAMPLES

#include <sys/dma.h>
#define HOST_RAM_SIZE 0x8000

extern int coherent_io_enabled;
my_attach(id,isc)
PCI_ID id;
struct isc_table_type *isc;
{
caddr_t host_ram;
int i, pages_mapped;
caddr_t io_tmp, proc_tmp;
...
MALLOC(host_ram,cadr_t, HOST_RAM_SIZE,M_DYNAMIC,M_NOWAIT);
bzero(host_ram,HOST_RAM_SIZE);
pages_mapped = HOST_RAM_SIZE/NBPG;
if (HOST_RAM_SIZE % NBPG)
pages_mapped++;
if (!coherent_io_enabled) {
  /*
   ** need to ensure a contiguous
   ** buffer on processors that do
   ** not have an IO TLB, because
   ** our card expects contiguous
   ** space for task lists
   */
   proc_tmp = host_ram;
   io_tmp = wsio_get_pva(isc,proc_tmp);
   for (i=0;i < pages_mapped;
i++,io_tmp += NBPG,proc_tmp += NBPG) {
     if (io_tmp != wsio_get_pva(isc,proc_tmp)) {
       msg_printf("my_attach IO buffer not contiguous\n");
       FREE(host_ram,M_DYNAMIC);
       return(*my_pci_saved_attach)(id,isc)
     }
   }
}
...
return(*my_pci_saved_attach)(id,isc)

SEE ALSO
wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3),
wsio_unmap(WSIO3)
NAME

wsio_get_registers (WSIO3) – Get the register addresses of an interface card

SYNOPSIS

#include <sys/io.h>

caddr_t * wsio_get_registers (struct isc_table_type * isc);

PARAMETERS

isc A pointer to the ISC structure associated with an interface card.

DESCRIPTION

The wsio_get_registers() WSIO function retrieves the register addresses of the interface card associated with the given ISC entry.

Any module that has additional SPA space (for example, graphics) will need access to both of the register-property addresses. Normally, only the first register is retrieved and held in the ISC structure. This routine will allow drivers to get both register pointers if they need them. wsio_get_registers() is not supported for PCI interface cards.

RETURN VALUES

<>-1 Successful completion. The value is a pointer to an array of register sets for this module (for most modules, this will be an array of 1 or 2 elements, corresponding to HPA and SPA).

-1 Failure. isc is NULL or there was a problem retrieving the node’s registers property.

CONSTRAINTS
NAME

`wsio_install_driver` (WSIO3) - Install a driver's header structure into the WSIO CDIO.

SYNOPSIS

```c
int wsio_install_driver (wsio_drv_info_t * wsio_drv_info);
```

PARAMETERS

`wsio_drv_info` Pointer to the driver's `wsio_info_t` structure.

DESCRIPTION

The `wsio_install_driver()` WSIO function installs a driver's header structure into the WSIO CDIO.

RETURN VALUES

`wsio_install_driver()` returns the following values:

- 1  
   Successful completion.

- 0  
   Error. The major number specified for the driver is already in use. The following message is displayed on the system console and in the error-log file:

   ```
   wsio_install_driver:
   Install of driver driver failed.
   ```

CONSTRAINTS

EXAMPLES

```c
/* Declare the driver entry points */
static drv_ops_t  beep_ops = {
    beep_open, /* open */
    beep_close, /* close */
    NULL,       /* strategy */
    NULL,       /* dump */
    NULL,       /* psize */
};
```
NULL, /* mount */
NULL, /* read */
NULL, /* write */
beep_ioctl, /* ioctl */
NULL, /* select */
NULL, /* option_1 */
NULL, /* reserved1 */
NULL, /* reserved2 */
NULL, /* reserved3 */
NULL, /* reserved4 */
0 /* flag */
};

/* Declare the CDIO driver-specific fields */
/* Flags DRV_CHAR/DRV_BLOCK/DRV_PSEUDO/DRV_SCAN/DRV_MP_SAFE/
  DRV_SAFE_CONF */
static drv_info_t beep_info = {
  "beep", /* char *name for device type */
  "graf_pseudo", /* char *name for device class */
  DRV_PSEUDO|DRV_CHAR,/*ubit32 flags pseudo? block? char? scan? */
  -1,  /* int b_major maj dev# if block type */
  168, /* int c_major maj dev# if char type */
  NULL, /* struct cdio *cdio drivers set to NULL */
  NULL, /* void *gio_private drivers set to NULL */
  NULL /* void *cdio_private drivers set to NULL */
};

/* Declare the WSIO driver-specific fields */
static wsio_drv_data_t beep_data = {
  "hil", /* char *drv_path match probes-drivers */
  T_DEVICE, /* sbit8 drv_type type of H/W: dev or IF */
  DRV_CONVERGED,
  /* ubit32 drv_flags DRV_CONV... or NOT_C.... */
  NULL,
  /* int (*drv_minor_build)() minor# formatter */
  NULL
  /* int (*drv_minor_decode)() interpreter */
};

static wsio_drv_info_t beep_wsio_info = {
  &beep_info, /* drv_info_t *drv_info */
  &beep_ops, /* drv_ops_t *drv_ops driver entry points */
  &beep_data /* wsio_drv_data_t *drv_data */
};

beep_install()
{ /* register driver with WSIO and return any error */
return( wsio_install_driver( beep_wsio_info ) );
}

SEE ALSO

install_driver(CDIO3)
NAME

wsio ISC to instance (WSIO3) – Retrieve an instance number of an iotree node

SYNOPSIS

#include <sys/wsio.h>
#include <sys/ioparams.h>

int wsio ISC to instance (struct isc_table_type *isc,
 hw_path_t *dev_hw_path);

PARAMETERS

isc A pointer to the ISC structure associated with the
interface card for the device.

dev_hw_path A pointer to a structure containing device hardware
path information relative to the interface card, or NULL
if the card instance is desired.

DESCRIPTION

The wsio ISC to instance() WSIO function provides an instance
number of an iotree node that is a descendant of the card or device node
specified by the isc and dev hw path parameters, according to the
following rules:

• If isc is valid and dev hw path is NULL, the instance number of an
  iotree node corresponding to the interface card is returned.

• If isc is valid and dev hw path contains a path to a valid device
  (relative to the device's interface card), the instance number of the
  iotree node corresponding to the device is returned.

This function will not provide valid instance numbers if it is accessed
before the driver has actually claimed a device (that is, before the
driver's attach routine has successfully claimed and initialized an ISC
structure). Until that point, the driver is not associated with an iotree
node and will not have a valid instance number. Using this service within
a driver's driver_if_init() or driver_dev_init() routines and later
will yield a valid result.
RETURN VALUES

wsio_isc_to_instance() returns the following values:

>=0        Successful completion. The value is the matching instance number.
-1         Error.

CONSTRAINTS

SEE ALSO
NAME

wsio_map(WSIO3) - Map all or part of a host address range into an I/O virtual address range

SYNOPSIS

#include <sys/dma.h>

int wsio_map (struct isc_table_type *isc, io_map_t *map_cb, int hints, int range_type, struct iovec *host_range, struct iovec *io_range);

PARAMETERS

isc
A pointer to the isc struct for this device which contains information that is bus specific used by the underlying mapping services to correctly set up the mapping hardware and other bus specific details.

map_cb
A control structure which is private to the mapping service. This structure stores the mapping context across multiple calls to wsio_map(). map_cb can be a local variable. If used, it must be initialized before the first call to wsio_map() via a call to init_map_context(). A non-NULL value causes the default IOVA allocation scheme to be used regardless of previous calls to wsio_set_attributes().

hints
Hints which change the behavior of wsio_map():

IO_CONTIGUOUS Indicates that wsio_map() must allocate a single contiguous I/O virtual range. If wsio_map() is unable to do this, it will return -1. Of course, on noncoherent systems, the object must be physically contiguous. This hint implies IO_IGN_ALIGNMENT.

IO_IGN_ALIGNMENT Indicates that wsio_map() should not set the SAFE bit for cache-line
fragments and that `wsio_map()` should not return separate `io_ranges` for cache-line fragments.

**IO_LOCK**
Forces the LOCK bit to be set in the I/O PDIR for this mapping. This hint should be used for devices which can request exclusive access to memory. For instance, an EISA card can assert the EISA LOCK signal to request exclusive access to memory. Memory objects used in this way must be mapped with the `IO_LOCK` hint specified.

**IO_NO_SEQ**
Turns off the SEQUENTIAL bit in the I/O PDIR for this mapping. This inhibits prefetching of data for this object by the I/O Adapter. This hint has no effect for modules which do not prefetch data.

**IO_SAFE**
Forces the SAFE bit to be set in the I/O PDIR for this mapping. This causes the I/O subsystem to perform read-modify-write bus transactions for this mapping. This hint should be specified if sub-cacheline sized DMA will be used for the buffer.

**IO_SEMA**
Provides a hint that this memory object will be used as a semaphore.

**IO_UPDATE**
Forces the UPDATE bit to be set in the I/O PDIR for this mapping.

**range_type**
The type of host address for `host_range`. If `range_type > 0`, then this is the space ID of the address range. Other values are:

- `KERNELSPACE` Kernel virtual buffer (same as space ID = 0)

**host_range**
A pointer/length pair indicating the host address range of type `range_type`. The length will be modified by the service to indicate bytes remaining to be mapped. This
length will also be returned by `wsio_map()`.

`io_range` A pointer/length pair filled by `wsio_map()` with the I/O virtual address range or the EISA address range.

**DESCRIPTION**

The `wsio_map()` WSIO function maps the host address range into an I/O virtual address range. For EISA busses, the range is also mapped into EISA space. It may take multiple calls to `wsio_map()` to map the entire host range due to cache-line and page alignment restrictions. If the `host_range` is not aligned on a cache-line, then the first `io_range` will also not be cache-line aligned and will contain only the bytes in the same cache-line. Likewise, if the `host_range` does not end on a cache-line, then the last `io_range` will represent a cache-line fragment (but will be cache-line aligned). Buflets for the cache-line fragments must be managed by the caller.

Since the behavior and side effects are different for coherent and noncoherent systems, these are discussed separately below. Although the behaviors are slightly different, drivers should not have dependencies on these differences. In all cases, the programming model is the same. For each mapping:

- Call `init_map_context()` for the `map_cb` if used.
- Repeatedly call `wsio_map()` until the return value is less than or equal to 0.

All mappings remain valid until the `io_range` is unmapped via calls to `wsio_unmap()`. If the `io_range` is reused, `dma_sync()` must be used to resynchronize it.

**Behavior on Noncoherent Systems**

On noncoherent systems, the I/O virtual address is equivalent to the host physical address. `wsio_map()` will return an `io_range` at each page break in the `host_range`. If the mapping is for an EISA module, then the `io_range` is filled with EISA addresses and the EISA map will contain the host physical address.

**Behavior on Coherent Systems**

On coherent systems, the I/O virtual address is obtained by creating a mapping in the I/O PDIR. For cache-line fragments, the `SAFE` bit will be set in the I/O PDIR unless inhibited by `IO_IGN_ALIGNMENT`. However, it is expected that buflets will be used by the caller to handle them. If the mapping is for an EISA module, then the `io_range` is filled with EISA...
addresses and the EISA map will contain the corresponding IOVs.

**RETURN VALUES**

wsio_map() returns one of the following values:

- >0  The number of bytes remaining in the host_range.
- 0   The range was fully mapped.
- -1  The necessary resources could not be obtained.

**CONSTRAINTS**

**WARNING**

Under certain conditions a PCI master MUST have the IO.Safe and IO.NoSeq flag bits set to ensure coherency. See pci-errata(PCI5).

**EXAMPLES**

The following function maps a set of host pointer/length pairs given by host_vec (WSIO driver):

```c
int
my_driver_output(struct isc_table_type *isc, int vec_cnt, struct iovec *host_vec)
{
    io_map_t context;
    struct iovec *io_vec;
    struct my_dma_type *dma_desc;
    int resid;

    init_map_context(&context);

    /* Allocate a DMA structure for my DMA model */
    MALLOC( dma_desc, sizeof(struct my_dma_type) );

    /* Synchronize DMA buffer (outbound DMA) */
    dma_sync_IO(KERNELSPACE, host_vec->iov_base,
                host_vec->iov_len, IO_SYNC_FORDEV);

    /* Point to the beginning of the DMA Vector area */
```
io_vec = &dma_desc->iov;

/* Do mapping for each host vector */
for (; vec_cnt; host_vec++){
    do {
        resid = wsio_map(my_isc, context, 0, KERNELSPACE,
                         host_vec, io_vec);
        if (resid < 0){
            /* Handle Error condition */
        }

        /* Point to next DMA vector */
        io_vec++;
    } while (resid > 0);
}

dma_desc->iov_cnt = io_vec - &dma_desc->iov;

/* Kick off the DMA */
return my_start_output(isc, dma_desc);

The following example attempts to allocate 32Kbytes of contiguous memory for a PCI SCSI bus master’s task lists. Because there is no API for contiguous memory on a Noncoherent system the routine checks for this. This scheme has the highest chance of success if it is done during PCI attach time because memory is generally not fragmented at that time. For 10.20 coherent_io_enabled indicates whether this is a Coherent system or a Noncoherent system.

#define HOST_RAM_SIZE 0x8000
#define TMP_BUF_SIZE 0x200
extern int coherent_io_enabled;

caddr_t tmp_buf, host_ram;
caddr_t tmp_buf_phys, host_ram_phys;

sample_pci_attach(id, isc)
PCI_ID id;
struct isc_table_type *isc;
{
    caddr_t tmp_buf, host_ram;
    caddr_t tmp_buf_phys, host_ram_phys;
    struct iovec host_vec, io_vec;
    struct iovec *io_vec_ptr *workptr;
    io_map_t map_cb;

    /* code to check that it’s our card */
/* set up memory */
MALLOC(tmp_buf, caddr_t, TMP_BUF_SIZE, M_DYNAMIC, M_NOWAIT);
bzero(tmp_buf, TMP_BUF_SIZE);
MALLOC(host_ram, caddr_t, HOST_RAM_SIZE, M_DYNAMIC, M_NOWAIT);
bzero(host_ram, HOST_RAM_SIZE);

/*
 ** WSIO mapping services do different things
 ** on coherent IO systems (C-Class) and
 ** non-coherent IO systems (B-Class).
 **
 ** In both cases the mapping call manipulates
 ** the host iovec base and length fields so
 ** that wsio_map can be called repeatedly.
 **
 ** -----------------------------
 ** coherent IO systems behave as follows:
 ** A single call attempts to map all
 ** pages in host.iov_len. The IO_CONTIGUOUS
 ** flag forces a call failure if the mapping
 ** crosses a 'range' boundary (currently 32K)
 ** AND unmaps all of the pages. If the
 ** IO_CONTIGUOUS flag is not set, the call
 ** will return with 0, but with host.iov_len > 0,
 ** indicating that you need to call wsio_map
 ** again, n.b., the next mapping may NOT be
 ** contiguous from the point of view of PCI.
 **
 ** -----------------------------
 ** non-coherent IO systems behave as follows:
 ** The IO_CONTIGUOUS flag is ignored, and
 ** at most, each call will map a single
 ** page (4K), there is no guarantee that
 ** malloc will have given you a contiguous
 ** buffer. Basically, you need to check
 ** each page and call wsio_map for each page.
 **
 ** The unmap call is a no-op on B-Class
 **
 ** n.b., don't depend upon the 'range' boundary
 ** for future releases remaining at 32K. I went
 ** ahead and commented this because we have seen a
 ** failure mode due to a mapping order of:
 ** tmpbuf = 200 bytes = a page, and then
 ** hostram = 0x7a4 bytes = 8 pages
WSIO Reference Pages
Functions, Macros and Structures

** i.e., a total of 9 pages crossing a
** range boundary -> the map call failed.
** */

int pages_mapped = HOST_RAM_SIZE / NBPG;
int i, map_ret;

if (HOST_RAM_SIZE % NBPG)
    pages_mapped ++;
/* see if we have a physically contiguous buffer
* on B-Class
*/
if (!coherent_io_enabled) {
    caddr_t phys_tmp, virt_tmp;
    virt_tmp = host_ram;
    phys_tmp = wsio_get_pva(isc, virt_tmp);
    for (i = 0; i < pages_mapped; i++, phys_tmp += NBPG, virt_tmp += NBP)
    {
        if (phys_tmp != wsio_get_pva(isc, virt_tmp)) {
            msg_printf("sample attach B-Class buffer not contiguous\n"
        );
        FREE(tmp_buf, M_DYNAMIC);
        FREE(host_ram, M_DYNAMIC);
        return (*sample_pci_saved_attach)(id, isc);
    }
}
/*
** now do the mappings -
** do hostram first because
** it fills a C-Class `range`
*/
MALLOC(io_vec_ptr, struct iovec *,
       sizeof(struct iovec) * pages_mapped, M_DYNAMIC, M_NOWAIT);
if (io_vec_ptr == NULL) {
    FREE(tmp_buf, M_DYNAMIC);
    FREE(host_ram, M_DYNAMIC);
    return (*sample_pci_saved_attach)(idc, isc);
}
workptr = io_vec_ptr;
host_vec.iov_base = host_ram;
host_vec.iov_len = HOST_RAM_SIZE;
init_map_context(&map_cb);
for (i = 0; i < pages_mapped; i++, workptr++) {
map_ret = wsio_map(isc, &map_cb,
    IO_CONTIGUOUS | IO_SAFE | IO_LOCK,
    KERNELSPACE,&host_vec, workptr );
if (map_ret < 0) {
    msg_printf("sample attach: could not map hostram pointer\n" );
    FREE(tmp_buf, M_DYNAMIC);
    FREE(host_ram, M_DYNAMIC);
    FREE(iov_ec_ptr, M_DYNAMIC);
    return (*sample_pci_saved_attach)(id, isc);
}
if (i==0)
    host_ram_phys = io_vec_ptr->iov_base; /* base addr for PCI */
*/
if (map_ret == 0)
    break;
} /*
 ** now map the tmp_buf
 **
 ** Note that we map it after hostram
 ** because 10.20 uses a 32 K range and
 ** if we did it before hostram the
 ** contiguous call would fail on a C class
 ** because we would have used up TMP_BUF_SIZE
 ** of the 32 K range.
 */
init_map_context(&map_cb);
host_vec.iov_base = tmp_buf;
host_vec.iov_len = TMP_BUF_SIZE;
    if ( wsio_map (isc , &map_cb ,
        IO_CONTIGUOUS | IO_SAFE | IO_LOCK,
        KERNELSPACE , &host_vec , &io_vec ) ) {
        printf("sample attach Could not map() tmp_buf pointer.\n" );
    }
io_vec.iov_base = host_ram_phy_addr;
io_vec.iov_len = HOST_RAM_SIZE;
wsio_unmap(isc, io_vec);
FREE(tmp_buf, M_DYNAMIC);
FREE(host_ram, M_DYNAMIC);
FREE(io_vec_ptr, M_DYNAMIC);
return (*sample_pci_saved_attach)(id, isc);
} /* set up rest of stuff e.g., isrlink
isc_claim(isc, &wsio_sample_drv_info);
return (*sample_pci_saved_attach)(id, isc);
SEE ALSO

dma_sync(CDIO3), init_map_context(CDIO3), pci_errata(PCI5),
wsio_fastmap(WSIO3), wsio_remap(WSIO3),
wsio_set_attributes(WSIO3), wsio_unmap(WSIO3)
NAME

wsio_probe_dev_info(WSIO4) - WSIO device probe information

SYNOPSIS

#include <sys/wsio.h>

struct wsio_probe_dev_info {
    unsigned short instance;
    unsigned short target;
    unsigned short opt_1;
    unsigned short opt_2;
    unsigned short opt_3;
};

DESCRIPTION

The wsio_probe_dev_info structure contains some of the device file information needed for WSIO driver probe routines. This structure communicates hardware path information within driver probe routines (especially for those cases where drivers may split their probe routines into two routines: one for determining the next address to probe and one to actually try to build and open a device file for that address). The hardware addressing information in this format can easily be passed between two routines and is needed for building special device files for the devices being probed.

STRUCTURE MEMBERS

instance  Instance number of the nearest interface card ancestor.
target    Relative hardware address of first layer to be probed.
opt_1     Optional. Sometimes used for the hardware address of the second layer to be probed (e.g., LUN).
opt_2     Driver-discretionary element.
opt_3     Driver-discretionary element.
RETURN VALUES

CONSTRAINTS

EXAMPLE

A SCSI probe example might consists of two routines. `scsi_probe_function()` determines the next address to be probed and `scsi_probe()` builds device files and actually tries to open devices. Addressing information about the current node we're trying to probe is passed between the two routines via this structure. Elements of the structure are used by `scsi_probe()` to build device files as follows:

```c
dev = ((major_num << 24) & 0xff000000);
dev |= ((probe_dev->instance << 16) & 0x00ff0000);
dev |= ((probe_dev->target << 12) & 0x0000f000);
dev |= ((probe_dev->opt_1 << 8) & 0x00000f00);
if ( (sctl_open(dev)) == 0 )
{
    /* Do an ioctl() on the device to get the ID information
     * for building the name, description, and id strings.
     */
}
```

SEE ALSO

`wsio_register_probe_func(WSIO3)`
NAME

wsio_register_addr_probe(WSIO3) - Register a driver probe function.

SYNOPSIS

void wsio_register_addr_probe (int (*func)(), char *drv_name);

PARAMETERS

func A pointer to the driver probe function.

drv_name An ASCII string indicating the name of the driver.

DESCRIPTION

The WSIO service wsio_register_addr_probe() is used to register an interface driver’s probe function. The probe function is used by WSIO SCAN to look for I/O devices underneath interface cards claimed by the driver. The drv_name parameter must match the name field of the driver’s drv_info_t structure.

The driver probe function must have the following calling syntax:


drv_addr_probe( void *handle,
    int (*dev_probe)(),
    drv_info_t *drv_info,
    void *probe_id,
    hw_path_t *hw_path,
    struct isc_table_type *isc,
    int probe_type,
    char *name,
    char *desc )

handle A pointer to a GIO structure. Drivers should not touch this structure.

dev_probe A pointer to a probe function registered via the WSIO service wsio_register_dev_probe() if one exists, else NULL.

drv_info A pointer to the driver’s drv_info_t structure.

probe_id A unique identifier for the device found.

hw_path When an input, the hardware path of the last device
found. When an output, the hardware path of the next device to be found.

isc  A pointer to the *isc_table_type* structure of the interface card being probed.

probe_type  One of three types of probe, which are:

PROBE_FIRST  Find the first device underneath the interface card.

PROBE_NEXT  Find the next device after the previous one found.

PROBE_ADDRESS  Look for a device at the hardware address specified in *hw_path*.

name  A pointer to a string initialized with the device's name such as *scsi_disk*. This information is used to match the device to a driver based on the information in the *drv_path* field of the *wsio_drv_data_t* structure.

desc  A pointer to a string with a description of the device.

When the driver probe function is called with a *probe_type* of PROBE_FIRST the function should find the first device underneath the interface card specified by the *isc* parameter. The *hw_path* parameter has the address of the interface card. When the driver probe function is called with a *probe_type* of PROBE_NEXT, the driver should find the next device after the last device found. The address of the last device is passed in the *hw_path* parameter. The driver then updates the *hw_path* parameter with the address of the new device. Each time the probe function reports a device it should return the additional information of *probe_id*, *name* and *desc*. The *probe_id* is a unique identifier. The *name* string should match the *drv_path* field in the *wsio_drv_data_t* structure of the driver that controls the device. *desc* is an ASCII string describing the device.

A probe function registered via *wsio_register_addr_probe()* can be used as a standalone probe function or in conjunction with another probe function registered by the service *wsio_register_dev_probe()* in the latter case, the probe function registered via *wsio_registered_dev_probe()* is passed in as the second parameter to the first.

This method can be used by a driver stack where a device driver knows
the syntax to talk to certain devices and an interface driver would know
the range of addresses for a given I/O bus. The interactive driver would
register an address probe function via \texttt{wsio\_register\_addr\_probe()}
and the device driver would register its probe function via
\texttt{wsio\_register\_dev\_probe()}

If the probe function is used as a standalone probe function then a \texttt{NULL}
value is passed in as the second parameter. Most drivers need only
register a single probe function using the WSIO service
\texttt{wsio\_register\_dev\_probe()}.  

**RETURN VALUES**

None

**EXAMPLE**

```c
int mydrv_install()
{
    (void)wsio\_register\_addr\_probe(mydrv\_probe, \"mydrv\")
    return(wsio\_install\_driver(&mydrv\_wsio\_info));
}
```

**CONSTRAINTS**

**SEE ALSO**

drv\_info(CDIO4), wsio\_drv\_data(WSIO4),
wsio\_register\_dev\_probe(WSIO\_DRV),
NAME

wsio_register_dev_probe(WSIO3) - Register a driver probe function

SYNOPSIS

```
int wsio_register_dev_probe (int type, int(*func)(), char *drv_name);
```

PARAMETERS

- **type**: Indicates what driver data the third parameter should match to. Valid values are:
  - **IF_CLASS**: The third argument `drv_name` is to be matched with the `drv_path` field of the `wsio_drv_data_t` structure.
  - **DRV_NAME**: The third argument, `drv_name`, is to be matched with the name field of the `drv_info_t` structure.

- **func**: A pointer to the driver probe function.

- **drv_name**: An ASCII string indicating the name or class of the driver.

DESCRIPTION

The WSIO service `wsio_register_dev_probe()` is used to register a driver probe function. The driver probe function is used by WSIO SCAN to look for I/O devices beneath specific interface cards. Which cards to scan depend on the values of the first and third parameters. The third parameter, `drv_name`, is an ASCII string that is used to match the probe function to specific driver/interfaces cards. The first parameter, `type`, is used to indicate what driver information the ASCII string is to be matched to. If the parameter has the value **IF_CLASS**, it indicates the string should be matched to the `drv_path` field of the driver's `wsio_drv_data_t` structure. If the `type` parameter is set to the value of **DRV_NAME**, the third argument is matched with the name field of the driver's `drv_info_t` structure.

A value of **DRV_NAME** causes a tight pairing of the probe function to a particular driver since the probe is matched to the driver's name. A value of **IF_CLASS** is more general since several drivers may have the same
Probe functions registered via the service `wsio_register_dev_probe()` should have the following calling syntax:

```c
drv_probe( void *handle,
        drv_info_t *drv_info,
        void *probe_id,
        hw_path_t *hw_path,
        struct isc_table_type *isc,
        int probe_type,
        char *name,
        char *desc )
```

- **handle**: A pointer to an internal GIO structure. Drivers should not attempt to access it.
- **drv_info**: A pointer to the `drv_info_t` structure.
- **probe_id**: A unique identifier for the device found.
- **hw_path**: When an input, the hardware path of the last device found. When an output, the hardware path of the next device to be found.
- **isc**: A pointer to the `isc_table_type` structure of the interface card being probed.
- **probe_type**: The type of probe. The following types are supported:
  - `PROBE_FIRST`: Find the first device underneath the interface card.
  - `PROBE_NEXT`: Find the next device after the previous one found as indicated by the `hw_path` parameter.
  - `PROBE_ADDRESS`: Look for a device at the specific hardware address.
- **name**: A pointer to a string initialized with the device's name such as `scsi_disk`. This information is used to match the device to a driver on the information in the `drv_path`.
- **desc**: A pointer to a string with the device description. This is driver dependent.

When the driver probe function is called with a `probe_type` of...
PROBE_FIRST the function should find the first device underneath the interface card specified by the isc parameter. The hw_path parameter has the address of the interface card. When the driver probe function is called with a probe_type of PROBE_NEXT the driver should find the next device after the last device found. The address of the last device is the last element of the hw_path parameter. The driver then updates the hw_path with the address of the new device. Each time the probe function reports a device it should return the additional information of probe_id, name and desc. The probe_id is a unique identifier. The name string should match the drv_path field in the wsio_drv_data structure of the driver that controls the device. desc is an ASCII string describing the device.

RETURN VALUES

wsio_register_dev_probe() returns the following values:

0 Successful completion.
-1 Error.

EXAMPLE

int mydrv_install()
{
    (void)wsio_register_dev_probe(DRV_NAME,
        mydrv_probe, "mydrv");

    return(wsio_install_driver(&mydrv_wsio_info));
}

CONSTRAINTS

SEE ALSO

drv_info(CDIO4), wsio_drv_data(WSIO4),
wsio_register_addr_probe(WSIO_DRV),
NAME

wsio_register_probe_func(WSIO3) - Insert a driver-specified probe function into the global probe list

SYNOPSIS

int wsio_register_probe_func (int (* func)(()), char * if_class);

PARAMETERS

func The name of your probe function.
if_class The string that will be matched with the drv_path field of driver wsio_drv_data structs registered during driver_install().

DESCRIPTION

The wsio_register_probe_func() WSIO function inserts a driver-specified probe function into the global probe list. The underlying probe routines match your if_class string with driver drv_path strings to determine which probe to use. These routines match, character by character, up to a space or underline character, at which point the match succeeds. Note that the maximum string length is limited to 16 characters. For example, if scsi was passed in as the if_class parameter when registering your probe function, it would successfully match scsi_ctl in the drv_path field of the scsi_ctl driver's wsio_drv_data struct.

RETURN VALUES

wsio_register_probe_func() returns an indeterminate value. Treat it as a void function.

CONSTRAINTS

SEE ALSO

driver_class_probe(WSIO_DRV), driver_addr_probe(WSIO_DRV),
wsio_drv_data (WSIO4)
NAME

wsio_remap (WSIO3) – Map a host range into a pre-mapped I/O Virtual Address range

SYNOPSIS

#include <sys/dma.h>

int wsio_remap (struct isc_table_type *isc, int range_type, 
               struct iovec *host_range, struct iovec *io_range);

PARAMETERS

isc An isc_table_type structure used to get interface and CDIO information.

range_type The type of host address for host_range. If range_type > 0 then this is the space ID of the address range. Other values are:

KERNELSPACE Kernel virtual buffer (same as space ID = 0)

host_range A pointer/length pair, indicating the host address range of type range_type. The length will be modified by the service to indicate bytes remaining to be mapped.

io_range A pointer/length pair, which was filled by a previous call to wsio_map(), wsio_fastmap(), or wsio_remap(). wsio_remap() will map the new host_range into this range.

DESCRIPTION

The wsio_remap() WSIO function is normally used by CDIOs such as EISA which have to manage CDIO specific map registers. It is generally not used by driver writers.

The wsio_remap() WSIO function maps a pre-allocated I/O virtual address to new host_ranges. The io_range must use exactly the same number of mapping resources as the previous mapping. This can be ensured by making sure the buffers are page-aligned and equal sizes.
Cache-line fragments are ignored (IO_IGN_ALIGNMENT is assumed).

Since the behavior and side effects are different for coherent and noncoherent systems, these are discussed separately below. Although the behaviors are slightly different, drivers should not have dependencies on these differences.

All mappings remain valid until the io_range is unmapped via calls to wsio_unmap(). If the io_range is reused, dma_sync_IO() must be used to resynchronize it.

Behavior on Noncoherent Systems

On noncoherent systems, the I/O virtual address is equivalent to the host physical address. This will likely not be the same as the I/O virtual address provided by the caller. Therefore, wsio_remap() will fill io_range with the new address range. If the mapping is for an EISA module, then the io_range is filled with EISA addresses and the EISA map will contain the host physical address.

Behavior on Coherent Systems

On coherent systems, the I/O virtual address is remapped to point to the new host_range. The page type bits are not modified.

RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The range was fully mapped.</td>
</tr>
<tr>
<td>-1</td>
<td>The necessary resources could not be obtained.</td>
</tr>
</tbody>
</table>

CONSTRAINTS

SEE ALSO
dma_sync(CDIO3), wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3), wsio_unmap(WSIO3)
NAME

wsio_set_attributes (WSIO3) - Set map function attributes

SYNOPSIS

#include <sys/dma.h>

void wsio_set_attributes (struct isc_table_type *isc, int attributes);

PARAMETERS

isc Pointer to an ISC structure.

attributes Attributes which change the default behavior of
wsio_map(), wsio_fastmap(), wsio_remap(), and
wsio_unmap(). More than one attribute can be set by
ORing them together. The following attributes are
defined:

IO_DEBUG_DMA Turns on additional checks in
mapping services. This should be
used for debugging only. The services
will call panic() if any problems are
detected.

IO_INTERLEAVED_DMA The device is likely to
interleave many I/O requests. Mass
storage devices are an example of this
type of device. This is the default
behavior.

IO_NONINTERLEAVED_DMA The device is likely to
satisfy a single I/O request at a time.
This is typical of networking devices.
This attribute cannot be specified
with IO_INTERLEAVED_DMA.

Even if this attribute is set, the
services will behave as if they are
IO_INTERLEAVED_DMA if the map_cb
argument is non-NULL in calls to
wsio_map(). Networking cards are
typical of noninterleaved devices.
DESCRIPTION

The `wsio_set_attributes()` WSIO function alters the default behavior of `wsio_fastmap()`, `wsio_map()`, `wsio_remap()`, and `wsio_unmap()`.

RETURN VALUES

`wsio_set_attributes()` is a `void` function.

CONSTRAINTS

SEE ALSO

`panic(KER2)`, `wsio_fastmap(WSIO3)`, `wsio_map(WSIO3)`, `wsio_remap(WSIO3)`, `wsio_unmap(WSIO3)`
NAME

wsio_uninstall_driver(WSIO3) - Uninstall a driver's header structure from the WSIO CDIO.

SYNOPSIS

int wsio_uninstall_driver (wsio_drv_info_t * wsio_drv_info);

PARAMETERS

wsio_drv_info Pointer to the driver's wsio_info_t structure.

DESCRIPTION

The wsio_uninstall_driver() WSIO function uninstalls a driver's header structure from the WSIO CDIO. wsio_uninstall_driver() is called by the driver prior to unloading.

RETURN VALUES

wsio_uninstall_driver() returns the following values

0 Successful completion.
<>0 Error.

CONSTRAINTS

SEE ALSO

wsio_install_driver(WSIO3)
NAME

wsio_unmap (WSIO3) – Function to unmap an I/O virtual address range

SYNOPSIS

#include <sys/dma.h>

void wsio_unmap (struct isc_table_type *isc, struct iovec *io_range);

PARAMETERS

isc A pointer to the isc struct holding bus-specific information used by the mapping services.

io_range A pointer/length pair, representing the I/O virtual range to be unmapped.

DESCRIPTION

On coherent systems, the resources associated with the mapping are released. On noncoherent systems, this function does nothing. In addition to the wsio_unmap() WSIO function, the caller must call dma_sync() during post-DMA cleanup for inbound data.

When multiple objects (I/Os) are mapped with a single map context, wsio_unmap() must not be called for ANY of the mapped objects until ALL the I/Os for that context have completed. Failure to ensure that all I/Os have completed may result in data corruption.

RETURN VALUES

wsio_unmap() does not return a value.

CONSTRAINTS

EXAMPLE

The following function cleans up after an inbound DMA:

#define SYNC() dma_sync(0,0,0,0)
void my_inbound_dma_cleanup(isc, vec_cnt, host_vec, dma_desc)
    struct isc_table_type *isc;
    int vec_cnt;
    struct iovec *host_vec;
    struct my_dma_type *dma_desc;
{
    struct iovec *io_vec;
    int dma_cnt;

    /* Point to the beginning of the DMA Vector area */
    io_vec = &dma_desc->iov;

    /* Unmap each DMA vector */
    for (dma_cnt=dma_desc->iov_cnt; dma_cnt>0; dma_cnt, 
     io_vec++)wsio_unmap(my_isc,io_vec);

    /* Inbound data so synchronize each host range */
    for (; vec_cnt; host_vec++)
        dma_sync(KERNLLSPACE,host_vec->iov_base, 
        host_vec->iov_len, 
        IO_SYNC_FORCPU|IO_NO_SYNC|IO_PREFETCHED);

    SYNC();
    FREE( dma_desc );
}

SEE ALSO

dma_sync(CDIO3), init_map_context(CDIO3),
wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3)
NAME

wsio_unregister_dev_probe(WSIO3) - Unregisters a driver probe function.

SYNOPSIS

int wsio_unregister_dev_probe (int type, char * name);

PARAMETERS

type Indicates what driver data the second parameter should be matched to. Valid values are:
- IF_CLASS The second argument, name is to be matched with the drv_path field of the wsio_dev_data_t structure.
- DRV_NAME The second argument, name is to be matched with the name field of the drv_info_t structure.

name An ASCII string indicating the name or class of the driver.

DESCRIPTION

The WSIO service wsio_unregister_dev_probe() is used to unregister a driver probe function that was previously registered by a call to wsio_register_dev_probe(). The type and name parameters passed to wsio_unregister_dev_probe() should be the same as the first and third arguments passed to wsio_register_dev_probe() when the driver registered the probe function.

The first parameter, type, is used to indicate what driver information the ASCII string is to be matched to. If the parameter has the value IF_CLASS, it indicates the string should be matched to the drv_path field of the driver's wsio_dev_data_t structure. If the type parameter is set to the value DRV_NAME, the second argument is matched with the name field of the driver's drv_info_t structure. The second parameter, name, is an ASCII string with the driver's name or path.

The service is used primarily by DLKM type drivers in their unload routines.
RETURN VALUES

wsio_unregister_dev_probe() returns the following values:

0  Successfully found and deleted the driver
-1  Not found

CONSTRAINTS

EXAMPLE

int mydrv_unload( void *arg)
{
    int ret;
    struct isc_table_type *isc;
    void (token, *priv_ptr;

    /*******************************************************************************/
    * Remove the attach function from the DLKM attach list
    /*******************************************************************************/
    if (mod_wsio_attach_list_remove (MOD_WSIO_CORE,
           &module_name_core_attach))
        return (ENXIO);

    /*******************************************************************************/
    * Unregister the device probe
    /*******************************************************************************/
    (void) wsio_unregister_dev_probe(IF_CLASS,"mydrv_path");
    "probe_name");

    /*******************************************************************************/
    * Uninstall the driver. If it fails, go back to the
    * load state and undo what has been done in the
    * unload routine.
    /*******************************************************************************/
    if(wsio_uninstall_driver(&module_name_wsio_info)) {
        return (ENXIO);
    }
    return(0);
}
SEE ALSO

wsio_drv_data(WSIO4), wsio_drv_info(CDI04),
wsio_register_dev_probe(WSIO_DRV)
5 Network Device Driver Reference Pages
This chapter contains manual reference pages for the data structures, kernel support routines, and macros essential for HP-UX networking device drivers.

The following data structures are used by the network interface layer:

- **hw_ift** Defined in /usr/conf/sio/lan_dlpikrn.h. See hw_ift(NET4).
- **hw_dlpi** Defined in /usr/conf/sio/lan_dlpikrn.h. See hw_dlpi(NET4).

Each device driver may maintain its `hw_ift_t` and `hw_dlpi_t` structure as part of a larger structure, the driver control block. The driver control block provides information used in driving and controlling the interface hardware.

The other reference pages describe the routines and macros for use specifically by networking device drivers. Each networking driver may use any or most of the routines and macros on the following reference pages, as well as other routines explained in other sections of this manual.
NAME

ETHER_MAP_IP_MULTICAST – Macro to translate IP multicast address to physical multicast address

SYNOPSIS

#include <netinet/if_ether.h>

void ETHER_MAP_IP_MULTICAST (struct in_addr * ipaddr,
                              u_char enaddr[6]);

PARAMETERS

ipaddr Pointer to the IP multicast address.
enaddr Returned pointer to a 6-byte physical address.

DESCRIPTION

The ETHER_MAP_IP_MULTICAST() network function is a macro that maps an IP multicast address to a 6-byte physical multicast address. It may be used by the Ethernet/802.3 and FDDI protocols. The high-order 25 bits of the physical address are statically assigned. The low-order 23 bits are taken from the low-order bits of the IP address.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO
NAME

hw_ift(NET4) Network Structure – hw_ift – networking commands support structure

SYNOPSIS

#include "/usr/conf/sio/lan_dlpikrn.h"

typedef struct hw_ift
{
    hw_dlpi_t   hp_dlpi;
    u_int      mac_type;
    u_int      llc_flags;
    u_int      mjr_num;
    u_int      nm_id;
    u_int      instance_num;
    u_int      mtu;
    char   *name;
    u_char    hdw_path[MAX_HDW_PATH_LEN];
    u_int      hdw_state;
    u_int      mac_addr_len;
    u_char    mac_addr[MAX_MAC_ADDR_LEN];
    u_int      features;
;/* This is placeholder for future features which
*DLPI may support. */
    uint8_t   *arpmod_name;

    uint32_t   ppa;
    uint32_t   watch_timer;    u_int reserved1;
;/* For Internal use only. */
    lock_t    *hwift_lock;
;/* MP protection. */
    struct hw_ift  *next;
} hw_ift_t;

DESCRIPTION

The hw_ift network structure provides a consistent method for the network system utilities, lanscan and lanadmin, to display detailed information on all network devices. (For information on the HP-UX local loopback diagnostic, refer to loopback(1M).)
The `hw_ift` structure also contains the `hp_dlpi` structure interface for support between HP-UX DLPI and device drivers.

**STRUCTURE**

- **hp_dlpi**
  It must be initialized to all zeros by using `bzero()`.

- **mac_type**
  A network media device type defined in `/usr/conf/sio/lan_dlpikrn.h`:
  - `DEV_8023`  IEEE 802.3 device.
  - `DEV_8025`  IEEE 802.5 device.
  - `DEV_ATM`  ATM device.
  - `DEVETHER`  Ethernet device.
  - `DEV_FDDI`  Fibre Channel device.

- **llc_flags**
  Link Level Control (LLC) encapsulation method. The flag values defined in `/usr/conf/sio/lan_dlpikrn.h` and listed below are used by the `hw_ift` structure to indicate the protocol type and encapsulation method.
  - `ETHERTYPE`  Ethernet type.
  - `IEEE`  IEEE 802.2 type.
  - `NOVELL`  Novell packet type.
  - `SNAP`  SNAP type.

- **mjr_num**
  Major number of the device file associated with the device. The major number could be statically assigned; if it is set to -1 the major number is allocated dynamically during driver initialization.

- **nm_id**
  Network management ID. `nm_id` should be initialized via a call to the `get_nmid()` routine.

- **instance_num**
  Device instance number. This number is in the value returned by calling the `wsio_isc_to_instance()` routine.

  Note: The `wsio_isc_to_instance()` routine cannot be called from a `driver_attach()` routine, but it can be called from a `driver_init()` routine.
**mtu**

Maximum transmission unit (MTU) size (number of bytes) for the particular type of link or encapsulation. The following are the MTU values (maximum transmission unit without header) and the defining header files for particular protocols used in type 1 connectionless communication.

### Table 5-1: MTU Size and Defining Header Files for Particular Protocols

<table>
<thead>
<tr>
<th>mtu Value</th>
<th>Header File</th>
<th>Maximum Packet Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETHERMTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>1500 bytes, Ethernet</td>
</tr>
<tr>
<td>FDDI_MTU</td>
<td><code>&lt;netinet/if_ethernet/if_ether.h&gt;</code></td>
<td>4352 bytes, SNAP for FDDI</td>
</tr>
<tr>
<td>IEEE8023_MTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>1497 bytes, IEEE 802.3</td>
</tr>
<tr>
<td>IEEE8025_16_MTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>4170 bytes, 16 Mb Token Ring</td>
</tr>
<tr>
<td>IEEE8025_4_MTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>4170 bytes, 4 Mb Token Ring</td>
</tr>
<tr>
<td>SNAP8023_MTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>1492 bytes, SNAP 802.3</td>
</tr>
<tr>
<td>SNAP8025_16_MTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>4170 bytes, SNAP for 16 Mb Token Ring</td>
</tr>
<tr>
<td>SNAP8025_4_MTU</td>
<td><code>&lt;netinet/if_ether.h&gt;</code></td>
<td>4170 bytes, SNAP for 4 Mb Token Ring</td>
</tr>
</tbody>
</table>

**NOTE**

The `<netinet/if_ether.h>` header file was not delivered in Release 10.20. You can obtain a copy of the file by contacting the Interface Program at e-mail address `interface@fc.hp.com`.

**name**

Driver device name that is used for naming shared libraries for `lanscan` and `lanadmin`.

**hdw_path**

Hardware path, which can be accessed by calling
Network Device Driver Reference Pages
Network Macros, Structures, and Functions

io_node_to_hw_path followed by
io_hw_path_to_str.

hdw_state Hardware state of the device: 0 if the device is OK. If
the device is not available, hdw_state must be set to
LAN_DEAD.

mac_addr_len Number of bytes of mac_addr for MAC address.

mac_addr MAC address of the device.

features Features supported by device. Six flags are supported:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRV_MBLK</td>
<td>This flag must be set since the third party network driver is purely based on STREAMS model.</td>
</tr>
<tr>
<td>DRV_MP</td>
<td>Set if the device driver is MP-scalable; that is, it uses spinlock() and spinunlock() to avoid race conditions. With this flag set, the driver cannot use any splN() calls.</td>
</tr>
<tr>
<td>DRV_IP_MULTICAST</td>
<td>This flag must be set if driver supports IP multicast feature.</td>
</tr>
<tr>
<td>DRV_LANC_PROMISC_SUPPORT</td>
<td>This flag must be set if driver supports promiscuous listening.</td>
</tr>
<tr>
<td>DRV_NO_FAST_PATH</td>
<td>This flag must be set if driver does not support fast path as described in “Transmission of Message Blocks”.</td>
</tr>
<tr>
<td>DRV_CKO</td>
<td>This flag must be set if driver supports TCP or UDP checksum calculations in hardware.</td>
</tr>
</tbody>
</table>

arpmod_name The name of ARP streams helper module. This helper module complements the generic ARP module to resolve addresses in networks such as Token Ring and Fibre Channel.

ppa PPA number for the interface. The driver should initialize this field with hw_ift->instance_num.

watch_timer For Hewlett-Packard internal use only. This field must
be set to 0 for non-Hewlett-Packard devices.

reserved1  Hewlett-Packard internal use only. It must be set to 0 for a non-Hewlett-Packard device.

hwift_lock  Pointer to a hwift_lock spinlock structure to protect the hw_if structure. It is initialized in hw_ift_attach().

next  Pointer to the next hw_if structure in the list. This field is set by calling the hw_ift_attach() routine during device driver initialization.

SEE ALSO

bzero(KER2), driver_attach(WSIO_DRV),
driver_if_init(WSIO_DRV), hw_ift_attach(NET3),
<driver>admin(1M), lanscan(1M), loopback(1M), spinlock(KER2),
spinunlock(KER2), wsio_isc_to_instance(WSIO3)
NAME

hw_ift_attach(NET3) Network Function – Link the hw_ift structure to a global list of hw_ift structures of active interfaces

SYNOPSIS

hw_ift_attach (hw_ift_t *hw_ift_ptr);

DESCRIPTION

The hw_ift_attach() network function links the hw_ift structure to a global list of hw_ift structures of active interfaces.

It is usually executed in your driver_init() routine. It is safely called after the driver claims the ICS: isc_claim()

PARAMETERS

hw_ift_ptr Pointer to the associated hw_ift structure.

CONSTRAINTS

SEE ALSO

driver_if_init(WSIO_DRV), hw_ift(NET4)
NAME

HWIFT_LOCK(NET3), HWIFT_UNLOCK(NET3) Network Functions -
Acquire/release hwift_lock spinlock

#include "/usr/conf/sio/lan_dlpikrn.h"

void HWIFT_LOCK (lock_t lock);

void HWIFT_UNLOCK (lock_t lock);

PARAMETERS

lock Pointer to an allocated lock.

DESCRIPTION

The HWIFT_LOCK() and HWIFT_UNLOCK() network functions are macros
that acquire and release the hwift_lock spinlock to protect the hw_ift
structure fields.

One spinlock, the hwift_lock field in the hw_ift structure, is defined to
protect the mac_addr and MIB structures pointed to by mib_ptr in the
hw_dlpi structure.

NOTE

The hwift_lock spinlock is allocated and initialized by the
hw_ift_attach() routine. As a result, the HWIFT_LOCK() and
HWIFT_UNLOCK() macros are not available until the hw_ift_attach()
routine has been executed.

Any code that tries to change the hw_ift->mac_addr field or the MIB
structure pointed to by the hw_dlpi->mib_ptr field should be
encapsulated by the HWIFT_LOCK() and HWIFT_UNLOCK() macros.
HWIFT_LOCK() must also be used to acquire the hw_ift_lock in cases
where the dlpi_ioctl() request is not able to complete immediately.
RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

hw_ift(NET4), hw_ift_attach(NET3)
NAME

kget_log_instance(NET3) Network Function - Produce a unique number for use as a log instance

SYNOPSIS

#include <net_diag.h>

unsigned short kget_log_instance();

PARAMETERS

None.

DESCRIPTION

The kget_log_instance() network function returns a unique number tied to the specific instance of a networking interface. This number is for use as a log instance value, for threading log messages together so all messages for the same instance can be identified together. A change in the log instance means that a new event is being logged.

The log instance value should be passed between subsystems through their interface parameter list, so each module may access it. If a module encounters a unique event, it obtains a log instance value. Otherwise, the module uses the current log instance value it was passed without calling kget_log_instance().

RETURN VALUES

kget_log_instance() returns the following values:

n A unique number for use as a log instance value.

CONSTRAINTS

SEE ALSO

klogg_write(NET3)
NAME

KLOG_CK(NET3) Network Function - Checks whether logging is enabled for the current subsystem.

SYNOPSIS

#include <sys/net_diag.h>
#include <sys/subsys_id.h>

int KLOG_CK (int subsys_id, int log_class);

PARAMETERS

subsys_id The unique ID number (assigned by Hewlett-Packard) of the calling subsystem.

log_class Defines the classification of event. All classes are defined in the header file <sys/subsys_id.h>. Four classes are defined for logging messages:

- INFORMATIVE Normal messages only.
- WARNING Warning messages.
- ERROR Error condition messages.
- DISASTER Critical error messages.

DESCRIPTION

The KLOG_CK() network function is a macro that allows the calling process to find out whether logging is enabled for the current subsystem.

RETURN VALUES

KLOG_CK() returns the following values:

- 0 Logging is disabled.
- 1 Logging is enabled.

CONSTRAINTS
SEE ALSO

klogg_write(NET3)
NAME

klogg_write(NET3) Network Function – Send log messages to the kernel trace and log facility

SYNOPSIS

```
#include <net_diag.h>
#include <subsys_id.h>

int klogg_write (short subsys_id, int class, int device_id,
                 u_short log_instance, caddr_t tl_packet,
                 int tl_packet_cnt);
```

DESCRIPTION

The klogg_write() network function sends log messages to the kernel trace and log facility. Prefiltering is done at the time of the log call, and unwanted messages are dropped.

PARAMETERS

- **subsys_id** The unique ID (number assigned by Hewlett-Packard) of the calling subsystem.
- **class** The classification of event. All classes are defined in the header file `<sys/subsys_id.h>`. Four classes are defined for logging messages:
  - INFORMATIVE Normal messages only.
  - WARNING Warning messages.
  - ERROR Error condition messages.
  - DISASTER Critical error messages.
- **device_id** The device ID number (for example, if_unit) of the calling subsystem message. If this is a non-applicable parameter, pass in -1.
- **log_instance** A unique static number used to identify the thread of events attending an interface. If this is a non-applicable parameter, pass in -1.
tl_packet

Either a pointer to an mbuf chain or a pointer to a set of iovec structures as determined by tl_packet_cnt. This structure is immediately copied into an mbuf chain owned by the tracing and logging facilities, so the calling routine need not copy the data and then pass a pointer to the data.

tl_packet_cnt

If -1, tl_packet points to an mbuf chain. If the value is greater than 0, it is the number of iovec structures (as defined in <sys/uio.h>) that tl_packet points to.

RETURN VALUES

This routine always returns a 0.

CONSTRAINTS

SEE ALSO

KLOG_CK(NET3)
NAME

KTRC_CHECK(NET3) Network Function – Check whether tracing is enabled; verify tracing activation on a per-interface-device basis.

SYNOPSIS

#include <sys/net_diag.h>
#include <sys/subsys_id.h>

int KTRC_CHECK (int subsys_id, int trace_kind, int device_id);

PARAMETERS

subsys_id The unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard).

trace_kind The kind of trace. Available kinds are defined in the <sys/subsys_id.h> header file as follows:

ERROR_TRACE_BIT Error tracing mask

HDR_IN_BIT Inbound header tracing mask

HDR_OUT_BIT Outbound header tracing mask

LOGGING_TRACE_BIT Log call tracing mask

LOOP_BACK_BIT For loopback

PDU_IN_BIT Inbound PDU tracing mask

PDU_OUT_BIT Outbound PDU tracing mask

PROCEDURE_TRACE_BIT Procedure entry/exit trace

PTOP_BIT For point to point

device_id The device ID number (for example, the if_unit value of an ifnet structure). It can be used for filtering on a per-interface basis.
DESCRIPTION

The `KTRC_CHECK()` network function is a macro that verifies whether tracing is enabled for the current subsystem and device interface.

This macro allows tracing on a per-interface-device basis. It can reduce the impact of tracing on performance where link-level tracing is enabled and the system has more than one interface card installed.

RETURN VALUES

`KTRC_CHECK()` returns the following values:

- 0  Tracing is disabled.
- 1  Tracing is enabled.

CONSTRAINTS

EXAMPLES

A hypothetical driver named `lan2` might use this macro as follows:

```c
if (KTRC_CHECK(NS_LS_LAN0, PDU_OUT_BIT,
               (&i->lancift.is_if)->if_unit))
{
    ktrc_write(...);
}
```

SEE ALSO

`ktrc_write(NET3)`
**NAME**

`ktrc_write(NET3)` Network Function - Send trace messages to kernel trace and log facility

**SYNOPSIS**

```c
#include <net_diag.h>
#include <subsys_diag.h>

int ktrc_write (short subsys_id, u_signed kind, int path_id, 
                int device_id, caddr_t tl_packet, 
                int tl_packet_cnt);
```

**DESCRIPTION**

The `ktrc_write()` network function sends trace messages to the kernel trace and log facility. Prefiltering is done at the time of the trace call, and unwanted messages are dropped.

**PARAMETERS**

- `subsys_id` The unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard).
- `kind` The kind of trace. All kinds are defined in the header file `<sys/subsys_id.h>`. The following are the defined trace kind values. They can be ORed to produce the combination of trace kinds.
  - `ERROR_TRACE_BIT` Error tracing mask
  - `HDR_IN_BIT` In bound header tracing mask
  - `HDR_OUT_BIT` Outbound header tracing mask
  - `LOGGING_TRACE_BIT` Log call tracing mask
  - `LOOP_BACK_BIT` For loopback
  - `PDU_IN_BIT` Inbound PDU tracing mask
  - `PDU_OUT_BIT` Outbound PDU tracing mask
PROCEDURE_TRACE_BIT
    Procedure entry/exit trace

PTOP_BIT
    For point to point

STATE_TRACE_BIT
    State machine tracing mask

path_id
    The connection path on the host. If this is a nonapplicable parameter, pass in -1.

device_id
    The device ID number (for example, if_unit) of the calling subsystem message. If this is a nonapplicable parameter, pass in -1.

tl_packet
    Either a pointer to an mbuf chain or a pointer to a set of iovec structures as determined by tl_packet_cnt. The calling routine will pass a pointer (cast to caddr_t) to an mbuf chain or an iovec structure. This structure is immediately copied into an mbuf chain owned by tracing and logging facilities. Therefore, it is not necessary for the calling routine to copy the data and then pass a pointer to it.

tl_packet_cnt
    If -1, then tl_packet points to an mbuf chain. If greater than 0, this is the number of the iovec structure that tl_packet points to.

RETURN VALUES
    Always returns a 0.

CONSTRAINTS

SEE ALSO

KTRC_CHECK(NET3)
NAME

`set_up_8022(NET3)` Network Function - Set up global information for the filter and formatting functions

SYNOPSIS

```c
#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int set_up_8022 (u_char *buf_ptr, int len, u_char *dst_addr,
                 u_char *src_addr);
```

DESCRIPTION

The `set_up_8022()` network function sets up global information used by the filter and formatting functions `filter_packet()`, `format_link_nice()`, `format_link_raw()`, and `format_link_terse()`. This routine walks through the buffer, pointing to 802.2 data, and copies protocol header information to the appropriate global variables used by the filter and formatters.

Call this routine for each PDU_IN or PDU_OUT trace event.

PARAMETERS

- `buf_ptr` Pointer to the beginning of the 802.2 information. It should not include MAC info.
- `len` Length of the buffer, excluding the MAC header.
- `dst_addr` Pointer to the 6-byte destination MAC address, extracted by local methods from the MAC header.
- `src_addr` Pointer to the 6-byte source MAC address, extracted by local methods from the MAC header.

RETURN VALUES

Always returns 0.
CONTRAINTS

SEE ALSO

set_up_ether(NET3), set_up_ip(NET3), set_up_link(NET3)
NAME

set_up_ether(NET3) Network Function - Set up global information for the filter and formatting functions

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int set_up_ether ( u_char * buf_ptr, int len, u_char * dst_addr, u_char * src_addr, int ether_type);

DESCRIPTION

The set_up_ether() network function sets up global information used by the filter and formatting functions filter_packet(), format_link_nice(), format_link_raw(), and format_link_terse(). This routine walks through the buffer and copies protocol header information to the appropriate global variables used by the filter and formatter routines. Call this routine for each PDU_IN and PDU_OUT trace event that contains Ethernet packets.

PARAMETERS

buf_ptr Pointer to the beginning of the Ethernet data. It should not include the destination address, source address, or Ethernet type information.

len Length of the buffer, excluding destination, source, and Ethernet type.

dst_addr Pointer to the 6-byte destination MAC address, extracted by local methods from the MAC header.

src_addr Pointer to the 6-byte source MAC address, extracted by local methods from the MAC header.

ether_type Ethernet-type field from the MAC header.
RETURN VALUES
Always returns 0.

CONSTRAINTS

SEE ALSO
set_up_8022(NET3), set_up_ip(NET3), set_up_link(NET3)
NAME

set_up_ip(NET3) Network Function - Set up global information for the filter and formatting functions

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int set_up_ip (u_char * buf_ptr, int len);

DESCRIPTION

The set_up_ip() network function sets up global information used by the filter and formatting functions filter_packet(), format_link_nice(), format_link_raw(), and format_link_terse(). It walks through the buffer and copies protocol header information to the appropriate global variables used by the filter and formatters.

NOTE

Most link products should not use this routine. Call this routine only when no link information is available for output formatting (for example, NS_LOOPBACK).

PARAMETERS

buf_ptr Pointer to the beginning of the IP information, which should not include MAC information.

len Length of the buffer, excluding the MAC header.

RETURN VALUES

Always returns 0.

CONSTRAINTS
SEE ALSO

set_up_8022(NET3), set_up_ether(NET3), set_up_link(NET3)
NAME

set_up_link(NET3) Network Function – Set up global information for the link layer only

SYNOPSIS

```c
#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int set_up_link (u_char * buf_ptr, int len, u_char * dst_addr,
                 u_char * src_addr);
```

PARAMETERS

- **buf_ptr**: Pointer to beginning of the Data Link information. It should not include MAC info. The routine does not currently use this parameter, but it is here for future extensions.
- **len**: Length of the buffer, excluding the MAC header.
- **dst_addr**: Pointer to the 6-byte destination MAC address, extracted by local methods from the MAC header.
- **src_addr**: Pointer to the 6-byte source MAC address, extracted by local methods from the MAC header.

DESCRIPTION

The `set_up_link()` network function sets up global information only for the link layer and does not attempt to extract any upper layer information from the traced packet. It does the minimum setup necessary to use the `format_link_nice()`, `format_link_raw()`, and `format_link_terse()` functions.

NOTE

Use this routine only if the packet being formatted cannot be handled by `set_up_8022()`.
RETURN VALUES

Always returns 0.

CONSTRAINTS

SEE ALSO

set_up_8022(NET3), set_up_ether(NET3), set_up_ip(NET3)
NAME

subsys_N_format(NET_DRV) Network Function - Routine to format a single trace or log message from subsystem N.

SYNOPSIS

```
#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int subsys_N_format (ss_N_fmt_flag_type flags, 
                     char * binary-msg-ptr, 
                     char * options-ptr, int msg-cat-fd, 
                     int error-fd, int output-file-count, 
                     fp_result output-files[], 
                     char * time-buffer, int time-buffer-length, 
                     int print-op, int user-count, 
                     user_acct_result users[], err_num status);
```

DESCRIPTION

The subsys_N_format() network function is provided by the subsystem developer. It can have any unique name. You pass the name to Network Services by including it in a shared library that you specify in the nettlgen.conf configuration file. With this naming notification method, several subsystems can use the same subsys_N_format() function.

At run time, the netfmt command loads the library and calls the routine whenever data from the subsystem is encountered.

The subsys_N_format() network function formats a single trace or log message from the N subsystem. It may discard the message based on filter information in the options file defined by the subsys_N_get_options() function associated with the subsystem.

The successful integration of all subformatters for all subsystems requires that all subformatters follow prescribed guidelines. All subsystems should call tl_header_format1() for each binary message that they will be formatting.

Only the tl_format_fprintf(), tl_format_write(), and tl_raw_format() functions should be used to produce output. Only the
file descriptors and file pointers provided by the formatter may be used. No other files may be opened, and no user input may be solicited. No signals should be masked. The subformatter should never call `exit()`. The definitions of the various flags should be carefully followed. Subsystems should try to mirror the behavior of other subsystems as much as possible.

You are not permitted to use `printf()` or `write()` to produce output.

For example, to write messages to standard output, you would use the `tl_format_write()` function with the `output-files[0].fd` file descriptor. To write messages to standard error, you would use the `error-fd` file descriptor.

For simplicity and compatibility with anticipated future growth, use the following mechanism to access the information a subformatter requires. A subsystem can call the `tl_get_parms()` function anywhere within the subformatter to get a pointer to all of the information that a subformatter might need. Use this `tl_get_parms()` mechanism whenever possible, as explained further in `tl_get_parms(NET3)`.

All future parameter changes will be made through the `tl_get_parms()` function. For backwards compatibility, the old parameter list remains the same; but ignore it in favor of using the information returned by `tl_get_parms()`.

**PARAMETERS**

`flags` The type of flags is defined as:

```c
typedef struct
{
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;
```

- `verbosity_bit` When this bit is set, a high level of verbosity is selected (the default).
- `console_logging`
This bit is set if console logging is enabled, in which case the subformatter should only call the tl_header_format1() routine and provide very minimal additional information (to be kept to one line).

**highlight_bit** If this bit is set (the default) highlighted output is enabled.

**nice_mode_bit** This bit is set when nice formatting has been enabled (by default, this bit is not set). Nice formatting is the most descriptive mode of formatting. All possible information should be displayed in this mode of output. Nice mode is not usually used for log messages.

**terse_mode_bit** This bit is set when terse formatting has been enabled (by default, this bit is not set). Terse formatting should output only one line of output per trace record. Terse mode is not usually used for log messages.

**terse_link_mode_bit** If the terse_mode_bit is set, setting the terse_link_mode_bit should cause the link name to be included in the output.

**terse_time_mode_bit** If the terse_mode_bit is set, setting the terse_time_mode_bit should cause the timestamp to be included in the output.

**map_to_names_bit** This bit is set (the default) when numeric addresses should be resolved into names whenever possible. For example, an IP address should be displayed as a host name if the
A pointer to a buffer that contains the binary trace/log message to be formatted. The buffer contains the trace/log header, struct `tl_msg_hdr`, from the `<ntl.h>` file, followed by the trace/log data (from `ktrc_write()` or `klogg_write()`):

```c
typedef struct {
    unsigned short       hdr_len;
    short                subsystemid;
    int                  device_id;
    tl_msg_flag_type     flags;
    set_of_32            kind;
    set_of_32            class;
    set_of_32            version;
    int                  dropped_events;
    unsigned int         dropped_data;
    unsigned int         data_len;
    unsigned int         orig_data_len;
    struct timeval       time;
    int                  invoke_id;
    int                  path_id;
    unsigned short       log_instance;
    short                uid;
    unsigned int         connection_id;
} tl_msg_hdr_type;
```

**NOTE**
For tracing, the data may be truncated by the nettl command facilities. Check the `tl_msg_hdr->data_len` field to find out how much data was captured.

A pointer to a data structure defined by the subsystem for communication between the `subsys_N_get_options()` routine and the `subsys_N_format()` routine. If no options are used, this pointer is NULL. The actual type of the structure pointed to by `options.ptr` is entirely up to the subsystem developer.
msg-cat-fd  The file descriptor of the subsystem message catalog configured in nettlgen.conf. The formatter opens subsystem message catalogs using catopen(). The tl_check_cat_version() function can be used to check that the version of the message catalog corresponds to the version of the subformatter. Subsystems should not open their own message catalog files.

error-fd  A file descriptor that identifies the file that receives any fatal or nonfatal error messages (typically associated with stderr).

output-file-count  The number of output files to receive the formatted trace/log messages. It must be 1 for HP-UX.

output-files[]  An array of structures, each of which contains a file descriptor number, fd, and a result.

```c
typedef struct
{
    int fd;
    int result;
} fp_result;
```

The fd-designated output file, output-files[0].fd, receives the formatted trace/log messages. Only one output file is used for HP-UX; output-files[0].result is ignored. This output file will have been opened by the formatter driver.

time-buffer  A string containing the formatted time stamp from the trace/log header.

time-buffer-length  Length of the time-buffer string, not counting the null terminator.

print-op  For HP-UX, this parameter must be 0.

user-count  For HP-UX, this parameter must be 0.

users  For HP-UX, this parameter must be NULL.

status  Contains an error code value if the routine returns -1, indicating an error condition. <fmt.h> gives a complete
list of such error codes:

FMTERR_INV_FLAGS
   Invalid flags parameter.

FMTERR_INV_BIN_MP
   Invalid binary-msg ptr binary message pointer.

FMTERR_INV_OUT_FP
   Invalid output file pointer.

FMTERR_INV_MC_FP
   Invalid message catalog file pointer.

FMTERR_INV_TL_MSG
   Invalid trace/log message. The message is so corrupted that no formatting can be done.

FMTERR_SYS_ERROR
   An error has been returned from a system call.

RETURN VALUES

subsys_N_format() returns the following values:

0   Successful completion.

-1   Error. An appropriate error code is provided in the status field and an error message (if any) is given in the file named in error-fd.

CONSTRAINTS

SEE ALSO

catopen(3C), exit(2), klogg_write(NET3), ktrc_write(NET3),
subsys_N_get_options(NET_DRV), tl_check_cat_version(NET3),
tl_format_fprintf(NET3), tl_format_write(NET3),
tl_get_parms(NET3), tl_header_format1(NET3),
tl_raw_format(NET3)
NAME

subsys_N_get_options(NET_DRV) Network Function – Routine to process options for subsystem N

SYNOPSIS

#include <fmt.h>

int subsys_N_get_options (get_opt_parms_type * get-opt-parms-ptr);

DESCRIPTION

The subsys_N_format() network function is provided by the subsystem developer. It can have any unique name. You pass the name to Network Services by including it in a shared library that you specify in the nettlgen.conf configuration file. With this naming notification method, several subsystems can use the same subsys_N_get_options() function.

It is the responsibility of the subsys_N_get_options() routine to read the subsystem-specific options information from the filter command file and store any necessary information.

PARAMETERS

get-opt-parms-ptr

Pointer to a get_opt_parms_type structure that describes the subsystem.

STRUCTURE

The get_opt_parms_type structure is defined in <fmt.h> as:

typedef struct
{
    int *status_ptr;

}
FILE *subsys_strm;
FILE *error_strm;
FILE *log_strm;
int ss_id;
char *ss_name;
nl_catd ss_msg_cat;
get_opt_flag_type ss_n_get_opt_flag;
char **ss_options_ptr_ptr;
int ss_output_fd;
char *options_file_name;
} get_opt_parms_type;

status_ptr The error code indicating an error condition of the
routine if the returned value is -1; <fmt.h> gives a
complete list of such error codes:

FMTERR_INV_FLAGS Invalid ss_n_get_opt_flag
parameter.

FMTERR_NO_MEM There is not enough memory; a call
to malloc() failed.

FMTERR_INV_OPT_FD
Invalid options file descriptor passed in.

FMT_INV_MC_FD Invalid message catalog file
descriptor passed in.

FMTERR_SYS_ERROR
An error has been returned from a
system call.

subsys_strm A pointer to the file that refers to the temporary file
containing the options specifically for the N subsystem.
This file is created by the caller prior to invoking the
subsys_N_options() routine, and each line has been
converted to lowercase. All comments, blank lines, and
lines for other subsystems are already removed. In
addition, the keyword identifying this subsystem has
been stripped off each line, so only the options for this
particular subsystem are in the file. Due to a special
encoding of line number and other data, the
tl_get_line() routine must be used to get option
lines from this stream file.

error_strm A pointer to the file that is to receive error messages.
Network Device Driver Reference Pages
Network Macros, Structures, and Functions

log_strm
A pointer to the file that is to receive a summary of all options and files in effect for the subsystem, generated by the subsys_N_get_options() routine. The nettl command reports the contents of this file after all the subsystems have finished reading their respective filter command files.

ss_id
The subsystem ID number for the subsystem as found in the configuration file.

ss_name
The subsystem name for the subsystem as found in the configuration file.

ss_msg_cat
A file descriptor pointing to the message catalog for the subsystem as found in the configuration file.

ss_n_get_opt_flag
An option flag, defined as:

typedef struct
{
    u_int trace_log_bit: 1;
    u_int parse_only_bit: 1;
    u_int reserved: 30;
} get_opt_flag_type;

trace_log_bit
This flag is not needed and should not be used by subsys_N_get_options().

parse_only_bit
The flag is set when the subsys_N_get_options() routine does not need to process the information in the file, but only parse the input and check for syntax and semantic errors.

ss_options_ptr_ptr
A pointer to a pointer to a data structure containing the specific information processed by subsys_N_get_options() and passed on to subsys_N_format() to handle special formatting. This structure should be allocated and initialized by subsys_N_get_options().

ss_output_fd
A file descriptor referring to the file receiving the formatter output.
options_file_name A character string which contains the file name of the filter file passed to netfmt with the -c option. The file name can be used in error and warning messages produced by subsystem _N_get_options() while parsing the filter field. Subsequent messages need not display the file name.

RETURN VALUES

subsys_N_get_options() returns the following values:

0 Successful completion.
-1 Error. An appropriate error code is provided in the status_ptr field and an error message (if any) is given in the file named in error-strm.

CONSTRAINTS

SEE ALSO

malloc(3C), nettl(1M), subsys_N_format(NET_DRV), tl_get_line(NET3)
NAME

tl_banner_char(NET3) Network Function – Get the character used for
tl_header_format1() banner printing

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

char tl_banner_char (unsigned int kind_class);

PARAMETERS

kind_class The trace kind or log class of the message.

DESCRIPTION

The tl_banner_char() network function obtains the character to be
used when printing a header banner with the tl_header_format1() function. The character is based on the type of log class or trace kind.
This function helps to ensure consistent banners for all trace/log
messages.

RETURN VALUES

tl_banner_char() always returns a character to be used by
tl_header_format1().

CONSTRAINTS

SEE ALSO

tl_header_format1(NET3)
NAME

tl_check_cat_version(NET3) Network Function - Check compatibility between subsystem message catalog and subsystem formatter library

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int tl_check_cat_version (int msgcatfd, int setnum, int msgnum, char *expectedversion, FILE *errstream);

PARAMETERS

msgcatfd A file descriptor of the message catalog which contains the version string.
setnum The set number in the message catalog.
msgnum The message number in the message catalog.
expectedversion The version string that the message catalog is expected to contain.
errstream A FILE pointer to a stream that will receive error messages.

DESCRIPTION

The tl_check_cat_version() network function checks that the subsystem message catalog has a compatible version with the subsystem formatter library. If not, an appropriate warning message is issued.

RETURN VALUES

tl_check_cat_version() returns the following values:

0 Successful completion. The versions match.
-1 Error. The versions don't match or the file descriptor of
the message catalog is invalid.

CONSTRAINTS

SEE ALSO

t1_header_format1(NET3)
NAME

tl_format_fprintf(NET3) Network Function – Convert, format, and print arguments to standard output

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int tl_format_fprintf(FILE *stream, fmt_wrt_flag_type flags, error_num *status_ptr, char *format, [arg]...);

PARAMETERS

stream One of the FILE streams contained in the ss_N_fmt_parms_type structure returned by tl_get_parms().

flags Controls the output behavior of the tl_format_fprintf() routine. The value must be set before calling tl_format_fprintf().

typedef struct
{
    unsigned highlight : 1;
    unsigned wait_to_write : 1;
    unsigned reserved : 30;
} fmt_wrt_flag_type;

highlight Write the format data in inverse video.

wait_to_write Reserved for future use.

status_ptr Contains the error value if the routine returns a -1.

format The format character string contains two types of objects: plain characters that are copied to the output stream, and conversion specifications. Each string results in fetching 0 or more arguments, arg. The results are undefined if there are insufficient args for the format. If the format is exhausted while args
remain, the excess args are ignored.

arg Argument for the format character string.

DESCRIPTION

The \texttt{tl\_format\_fprintf()} network function converts, formats, and prints its arguments under control of the format. This routine behaves like \texttt{printf()} but must be used instead to give \texttt{netfmt()} control over the formatted buffer.

RETURN VALUES

\texttt{tl\_format\_fprintf()} returns the following values:

\begin{itemize}
\item [0] Successful completion.
\item [-1] Error.
\end{itemize}

Fatal errors are reported through the return value and the \texttt{status\_ptr} parameter. All error messages (as follows) are written to the file pointed to by the \texttt{error\_fd} parameter of the \texttt{subsys\_N\_format()} routine.

\begin{itemize}
\item \texttt{FMTERR\_FORMAT\_FPRTINF} An error occurred in writing to the output stream.
\item \texttt{FMTERR\_INV\_L\_STR} Invalid line pointer string.
\item \texttt{FMTERR\_INV\_OUT\_FD} Invalid output file descriptor.
\item \texttt{FMTERR\_SYS\_ERROR} An error has been returned from a system call within the \texttt{tl\_format\_write()} routine.
\end{itemize}

CONSTRAINTS

SEE ALSO

\begin{itemize}
\item \texttt{subsys\_N\_format(NET\_DRV)}, \texttt{tl\_format\_write(NET3)}, \texttt{tl\_get\_parms(NET3)}, \texttt{tl\_raw\_format(NET3)}
\end{itemize}
NAME

tl_format_write(NET3) Network Function – Write a buffer to standard output

SYNOPSIS

```c
#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int tl_format_write (u_char *input_line_ptr,
                     int input_line_byte_count, int error_fd,
                     fmt_wrt_flag_type flags, int output_file_count,
                     fd_result output_files[], int print_op,
                     int user_count, user_acct_result users[],
                     err_num *status_ptr);
```

PARAMETERS

**input_line_ptr** A character string that contains the message to be printed to the output files. `input_line_ptr` need not be null-terminated nor ended with a newline.

**input_line_byte_count** The byte count of the `input_line_ptr` message string.

**error_fd** A file descriptor pointing to a file to receive error messages from the `tl_format_write()` routine.

**flags** Controls output behavior of the `tl_format_write()` routine. The value must be set before calling `tl_format_write()`.

```plaintext
typedef struct
{
    unsigned int highlight : 1;
    unsigned int wait_to_write : 1;
    unsigned int reserved : 30;
} fmt_wrt_flag_type;
```

**highlight** Write the `input_line_ptr` data in inverse video.

**wait_to_write** Reserved for future use.
**DESCRIPTION**

The `tl_format_write()` network function writes a buffer to standard output, pointed to by `output_file[0].fd`. The buffer may be created by one or more calls to the `fprintf()` C library function.

**RETURN VALUES**

`tl_format_write()` returns the following values:

- **0**  
  Successful completion.

- **-1**  
  Error.

Fatal errors are reported through the return value and the `status_ptr` parameter. All error messages (as follows) are written to the file pointed to by `error_fd`:

- **FMTERR_FORMAT_WRITE**
  An error has occurred in writing to an output file.

- **FMTERR_INV_L_STR**
  Invalid line pointer string.

- **FMTERR_INV_OUT_FD**
  Invalid output file descriptor.
FMTERR_SYS_ERROR

An error has been returned from a system call within the `tl_format_write()` routine.

CONSTRAINTS

SEE ALSO

`tl_format_fprint(NET3), tl_raw_format(NET3)`
NAME

tl_get_line(NET3) Network Function – Obtain a line from a filter command file

SYNOPSIS

int tl_get_line (FILE *commandstream, char *line, int linesize, char *origline, int lineno, FILE *errorstream);

PARAMETERS

commandstream A FILE pointer that points to the temporary filter command file containing a single subsystem's filter commands; typically the subsys_strm field of the get_opt_parms_type parameter to subsys_N_get_options().

line The buffer where tl_get_line() stores the filter command line.

linesize Size of line (no more than 2048).

origline tl_get_line() routine will store the original filter command line as it appeared in the filter command file in origline.

lineno tl_get_line() will store the line number of origline in lineno as it appeared in the filter command file.

errorstream A FILE pointer to a stream that will receive error messages.

DESCRIPTION

The tl_get_line() network function obtains a line from a filter command file.

The core formatter reads the filter command file, collects the lines specific to a subsystem, then edits and stores them into a temporary file. It then calls your subsys_N_get_options() routine with a parameter set as a pointer to this temporary file. The subsys_N_get_options() routine must call the tl_get_line() routine to extract one line at a time from this temporary file for processing. The lines are upshifted, and redundant white space is removed. tl_get_line() will store the
"cleaned" filter command line in this buffer.

RETURN VALUES

tl_get_line() returns the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>0</td>
<td>End of file.</td>
</tr>
<tr>
<td>&lt;0</td>
<td>Error.</td>
</tr>
</tbody>
</table>

CONSTRAINTS

SEE ALSO

subsys_N_get_options(NET_DRV)
NAME

tl_get_parms(NET3) Network Function – Return a pointer to a
ss_N_fmt_parms_type data structure

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

ss_N_fmt_parms_type * tl_get_parms()

DESCRIPTION

The tl_get_parms() network function returns a pointer to a
ss_N_fmt_parms_type data structure containing members that a
subsystem subformatter needs in order to operate. The core formatter
builds and initializes this data structure before calling
subsys_N_format().

STRUCTURE

typedef struct
{
    int *ss_status_ptr;
    FILE *ss_output_strm;
    int ss_output_fd;
    FILE *ss_error_strm;
    int ss_error_fd;
    nl_catd ss_msg_cat;
    char *ss_name;
    char *ss_binary_msg_ptr;
    char *ss_options_ptr;
    ss_N_fmt_flag_type ss_n_fmt_flags;
    char *time_buffer;
    int time_buffer_length;
    int output_file_count;
    fd_result output_files[1];
    int print_op;
    int user_count;
    user_acct_result *users;
} ss_N_fmt_parms_type;
ss_status_ptr  Used by a subformatter to store an error code if it fails.

ss_output_strm  A FILE pointer that will receive the formatted trace/log message.

ss_output_fd  A file descriptor that will receive the formatted trace/log messages.

ss_error_strm  A FILE pointer that will receive any fatal or nonfatal error messages.

ss_error_fd  A file descriptor that will receive any fatal or nonfatal error messages.

ss_msg_cat  The subsystems’ message catalog descriptor to be used in catgets.

ss_name  A pointer to the subsystem name.

ss_binary_msg_ptr  A pointer to a buffer containing log/trace messages to be formatted.

ss_options_ptr  A pointer to a buffer containing information to be passed between the subsys_N_format() routine and the subsys_N_get_options() routine. See the options-ptr parameter in the subsys_N_format() routine.

ss_n_fmt_flags  Options flags: the ss_N_fmt_parms_type type is defined in <fmt.h> as:

```
typedef struct
{
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;
```

verbosity_bit  Set for verbose output.

console_logging  Set for console formatting.

highlight_bit  Set for highlighted output.
nice_mode_bit  Set for nice output.
terse_mode_bit  Set for terse output.
terse_link_mode_bit  Set for link messages in terse mode.
terse_time_mode_bit  Set for timestamp in terse mode.
map_to_names_bit  Reserved
time_buffer  A string containing the formatted timestamp from the trace/log header (see the time-buffer parameter of subsys_N_format()).
time_buffer_length  Length of the time_buffer string, not counting the null terminator.
output_file_count  The number of output files to receive the formatted trace/log messages. For HP-UX, this member must have a value of 1.
output_files[]  An array of structures consisting of a file descriptor and result variable for each file to receive the formatted trace/log output. For HP-UX, only output_file[0].fd refers to the file receiving the formatter output.
print_op  For HP-UX, this member has a value of 0.
user_count  For HP-UX, this member has a value of 0.
users  For HP-UX, this member has a value of NULL.

RETURN VALUES

tl_get_parms() returns the following values:

n  Successful completion. The value is a pointer to an ss_N_fmt_parms_type data structure containing members that a subsystem subformatter needs to operate.
CONSTRAINTS

SEE ALSO

```c
subsys_N_format (NET_DRV)
```
NAME

*tl_header_format1*(NET3) Network Function - Routine to format a single
trace or log header

SYNOPSIS

```c
#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int tl_header_format1 (char *header_ptr, int error_fd, 
                     ss_N_fmt_flag_type flags, 
                     char *kind_str, char banner_char, 
                     int output_file_count, 
                     fd_result output_files[], 
                     char *time_buffer, int time_buffer_length, 
                     int print_op, int user_count, 
                     user_acct_result users[], int location, 
                     err_num *status_ptr);
```

PARAMETERS

- **header_ptr** Points to a buffer that contains the header of the 
  trace/log message to be formatted.
- **error_fd** A file descriptor that refers to the file that will receive 
  any error messages.
- **flags** Option flags in an ss_N_fmt_flag_type structure, 
  defined in <fmt.h> and described in 
  tl_get_parms(NET3).
- **kind_str** A text message (typically the result of the 
  tl_log_class() or tl_trace_kind() function) to be 
  displayed for the kind field from the trace/log header. 
  This string must be null-terminated. The kind message 
  is truncated to 16 characters. If **kind_str** is NULL, the 
  kind field from the header is displayed as a decimal 
  value.
- **banner_char** The character to use in the banner header line 
  (typically the result of the tl_banner_char()
function). The subformatter may use this character to indicate differences in messages, such as inbound or outbound messages. For example, inbound messages could use the character "v" while outbound messages could use the character "^".

output_file_count
This is the number of output files to receive the formatted trace/log header output. For HP-UX, only one output file is used so this value is always 1.

output_files[]
An array of structures consisting of a file descriptor and a result variable for each file to receive the formatted trace/log header output. For HP-UX, only one output file is used: output_file[0].fd.

time_buffer
A string depicting the formatted time stamp from the trace/log header.

time_buffer_length
The length of time_buffer not counting the null terminator byte.

print_op
For HP-UX, this parameter must have a value of 0.

user_count
For HP-UX, this parameter must have a value of 0.

users
For HP-UX, this parameter must have a value of NULL.

location
A value that can be used to locate the source of the message in the code. This parameter is set by the subsystem and may be used to represent any information the subsystem desires.

status_ptr
The error value if the routine returns a -1:

FMTERR_INV_HDR
The trace/log header is invalid (corrupt).

FMTERR_INV_HDR_PTR
The trace/log header pointer is invalid.

FMTERR_INV_OUT_FD
The output file descriptor is invalid.

FMTERR_INV_MC_FD
The message catalog descriptor is invalid.

FMTERR_SYS_ERROR
An error was returned from a system call within `tl_header_format1()`.

**DESCRIPTION**

The `tl_header_format1()` network function formats a single trace or log header. The format of the output conforms to the standard HP-UX network tracing and logging recommendations. The formatted header is written to the output file specified by `output_file[0]. fd`. `tl_header_format1()` must be called by every subformatter after the subsystem filters have been processed. At a minimum, this may be the only output generated by the subformatter.

**RETURN VALUES**

`tl_header_format1()` returns the following values:

- 0   Successful completion.
- -1  Error. Fatal errors are reported through the `status_ptr` parameter.

All error messages are written to the file pointed to by `error_fd` parameter.

**CONSTRAINTS**

**SEE ALSO**

`tl_banner_char(NET3)`, `tl_get_parms(NET3)`, `tl_log_class(NET3)`, `tl_trace_kind(NET3)`
NAME

tl_log_class(NET3) Network Function – Return a text interpretation for a log class value

SYNOPSIS

char * tl_log_class (unsigned int class);

PARAMETERS

class The numeric log class of the message. The keywords are defined in <sys/subsys_id.h>.

<table>
<thead>
<tr>
<th>class</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INFORMATIVE</td>
</tr>
<tr>
<td>2</td>
<td>WARNING</td>
</tr>
<tr>
<td>4</td>
<td>ERROR</td>
</tr>
<tr>
<td>8</td>
<td>DISASTER</td>
</tr>
</tbody>
</table>

DESCRIPTION

The tl_log_class() network function returns a text interpretation of a log class. The log class is stored as an integer. This function converts that number into a string that can be used in the formatted output. For example, passing in a log class of 8 causes the return value to be DISASTER. The result of tl_log_class() is typically used as a parameter to tl_header_format1() when printing a header.

RETURN VALUES

tl_trace_kind() returns the following values:

- NULL A pointer to a text interpretation of a log class.
- NULL class is not a defined value.

CONSTRAINTS
SEE ALSO

tl_trace_kind(NET3)
NAME

tl_raw_format (NET3) Network Function - Format trace or log message into hexadecimal and printable ASCII characters

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int tl_raw_format (char * data_ptr, int num_bytes,
                   int start, int error_fd, raw_fmt_flag_type flags,
                   int output_file_count,
                   fd_result output_files[],
                   int print_op, int user_count,
                   user_acct_result users[], err_num* status_ptr);

PARAMETERS

data_ptr A pointer to the buffer that contains the data to be dumped in hexadecimal form.

num_bytes The number of bytes to dump from the buffer pointed to by data_ptr. There is no checking to ensure that the number of bytes given does not exceed the actual buffer length. If num_bytes is zero, no data will be dumped.

start The offset into the buffer pointed to by data_ptr where the dump should begin. If start is zero, the dump will begin at the byte pointed at by data_ptr.

error_fd A file descriptor that will receive error messages.

flags Reserved for future used; should be set to 0 by the caller.

output_file_count The number of output files to receive the raw dump. For HP-UX, this parameter must have a value of 1.

output_files An array of structures, each of which contains a file descriptor and a result code for the last operation on the file. For HP-UX, only one output file is used;
**DESCRIPTION**

The `tl_raw_format()` network function formats a trace or log message into both hexadecimal and printable ASCII characters. The raw formatted output appears as follows:

```
0:73 61 6d 70 6c 65 5f 6c 6f 67 5f 64 61 74 61 2e sample_log_data
16:20 6d 6f 72 65 5f 64 61 74 61 20 61 73 64 66 6a more_data asdfj
```

The left-most column gives the decimal byte offset. The center area is the hexadecimal display of the data. The right-most column is the printable ASCII display of the data. A period is displayed for any nonprinting character.

**RETURN VALUES**

`tl_raw_format()` returns the following values:

- **0** Successful completion.
- **-1** Error. Fatal errors are reported through the `status_ptr` parameter.

All error messages are written to the file pointed to by the `error_fd` parameter.

**CONSTRAINTS**

**SEE ALSO**

`tl_format_fprintf(NET3), tl_format_write(NET3)`
NAME

tl_trace_kind(NET3) Network Function - Return a text interpretation for a trace kind value

SYNOPSIS

#include <sys/subsys_id.h>

char * tl_trace_kind (unsigned int kind);

PARAMETERS

<table>
<thead>
<tr>
<th>kind Value and Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000000 HDR_IN_BIT</td>
<td>Inbound Protocol Header.</td>
</tr>
<tr>
<td>0x40000000 HDR_OUT_BIT</td>
<td>Outbound Protocol Header.</td>
</tr>
<tr>
<td>0x20000000 PDU_IN_BIT</td>
<td>Inbound Protocol Data Unit (including header and data).</td>
</tr>
<tr>
<td>0x10000000 PDU_OUT_BIT</td>
<td>Outbound Protocol Data Unit (including header and data).</td>
</tr>
<tr>
<td>0x08000000 PROCEDURE_TRACE_BIT</td>
<td>Procedure entry and exit.</td>
</tr>
<tr>
<td>0x04000000 STATE_TRACE_BIT</td>
<td>Protocol or connection states.</td>
</tr>
<tr>
<td>0x02000000 ERROR_TRACE_BIT</td>
<td>Invalid events or condition.</td>
</tr>
<tr>
<td>0x01000000 LOGGING_TRACE_BIT</td>
<td>Special kind of trace that contains a log message.</td>
</tr>
<tr>
<td>0x00800000 LOOP_BACK_BIT</td>
<td>Packets whose source and destination system are the same.</td>
</tr>
<tr>
<td>0x00400000 PTOP_BIT</td>
<td>Packets whose transmission is point to point.</td>
</tr>
</tbody>
</table>
DESCRIPTION

The `tl_trace_kind()` network function returns a text interpretation of a trace kind. The trace kind is stored as an integer. This function converts that number into a string that can be used in the formatted output. For example, passing in a trace kind of 0x80000000 causes the return value to be "HDR IN TRACE". The result of `tl_trace_kind()` is typically used as a parameter to `tl_header_format1()` when printing a header.

RETURN VALUES

`tl_trace_kind()` returns the following values:

- `NULL`: A pointer to a text interpretation of a trace kind.
- `NULL`: kind is not a defined value.

CONSTRAINTS

SEE ALSO

`tl_log_class(NET3)`
6 PCI Reference Pages
PCI Reference Pages
PCI Macros and Functions

PCI Macros and Functions

This chapter describes driver functions that are specific to PCI Services.

In the function synopses, each parameter type is prefixed with the comment /*IN*/ or /*OUT*/, indicating that the parameter is input to or output from the function. These comments are included here because they are helpful and because they are used as keywords on other platforms.

PCI Masters and Coherency

Be aware that certain combinations of WSIO mapping service calls can interact with PCI masters to create an inconsistent view of memory. This behavior is documented in pci_errata(PCI5).
NAME

CONNECT_INIT_ROUTINE(PCI3) PCI Macro – Associate an initialization routine with a driver

SYNOPSIS

```c
#include <sys/pci.h>

#define CONNECT_INIT_ROUTINE(isc, init_routine) 
((isc)->gfsw->init = (init_routine))
```

PARAMETERS

- `isc` Pointer to an ISC table associated with the device.
- `init` Pointer to your driver's initialization routine.

DESCRIPTION

CONNECT_INIT_ROUTINE() is a macro that associates a driver_if_init() routine with a driver. It is typically used in a device's driver_attach() routine. After all attach routines are executed, the driver_if_init() routine is called by the kernel to perform further card and driver initialization.

If the driver performs all needed initialization in its driver_attach() routine, a driver_if_init() routine is not needed.
PCI Reference Pages
PCI Macros and Functions

NAME

PCI_ATTACH_DEV_INIT_ERROR(PCI3) PCI Macro - Report initialization error to WSIO Services

SYNOPSIS

#include <sys/pci.h>

#define PCI_ATTACH_DEV_INIT_ERROR(isc) ((struct wsio_if_info *)((isc)->if_info)->flags |= INIT_ERROR)

PARAMETERS

isc Pointer to the ISC structure passed into your attach routine.

DESCRIPTION

PCI_ATTACH_DEV_INIT_ERROR() is a macro that reports to WSIO Services that an error occurred during the device's initialization. It causes an error flag to be set in a structure in isc. It should be used as appropriate in the device's driver_attach() or driver_if_init() routine.
NAME

pci_desc_bus_transactions_isc(PCI3) PCI Function – Describe the typical bus performance path transaction size

SYNOPSIS

#include <sys/pci.h>

int pci_desc_bus_transactions_isc (
    /*IN*/ struct isc_table_type * isc,
    /*IN*/ BUS_TRANS_DESC * desc);

PARAMETERS

isc Pointer to an ISC table associated with the device.
desc A pointer to a BUS_TRANS_DESC structure, defined as:

typedef struct bus_trans_desc {
    uint32_t read_width;
    uint32_t write_width;
    uint8_t reserved[20];
} BUS_TRANS_DESC;

where:

read_width The number of 32-bit words in the width of the read path. 0 means don't change the current value.
write_width The number of 32-bit words in the width of the write path. 0 means don't change the current value.
reserved Reserved for future extensions.

DESCRIPTION

The pci_desc_bus_transactions_isc() PCI function allows a driver to describe the bus transaction size of a card's typical performance DMA accesses. Its use is entirely optional, since PCI Services provides a reasonable, general-purpose default.
The key to understanding how to use this routine for performance tuning is to recognize that the PCI bus supports variable-length data transactions. These transaction lengths may not map directly to transaction lengths on other busses on the system. Also, performance depends on other busses initiating the appropriate transaction in advance.

The purpose of the routine is to provide a hint of the typical performance path transaction size used by a specific card. While PCI can technically support unlimited transfer sizes (specifically, a dynamic number of data phases per PCI transaction), most PCI device/functions have some preferred size or can be programmed to use a particular size. By providing this hint, the PCI Services can, for some bus adapters, set up the bus adapter hardware to better map cycles between busses.

If you don't use `pci_desc_bus_transactions_isc()`, PCI Services provide defaults that are intended to be safe and to give reasonable performance.

**RETURN VALUES**

`pci_desc_bus_transactions_isc()` returns one of the following values:

- **PCI_OKAY**: The hints were used.
- **PCI_BUS_HINTS_BAD_DATA**: The `BUS_TRANS_DESC` structure is incorrect.
- **PCI_BUS_HINTS_NOT_USED**: The hints are not supported in this configuration.

**CONSTRAINTS**
NAME

pci_get_fru_info_isc(PCI3) PCI Function – Get field-replaceable-unit (FRU) information for the device associated with an ISC

SYNOPSIS

```c
#include <sys/pci.h>

int
pci_get_fru_info_isc(
    /*IN*/ struct isc_table_type* isc,
    /*OUT*/ int* fru_info);
```

PARAMETERS

**isc** Pointer to an ISC table structure associated with the device.

**fru_info** A pointer to the location where the routine should place the FRU information.

DESCRIPTION

The `pci_get_fru_info_isc()` PCI function returns field-replaceable-unit (FRU) information for the device associated with an ISC.

The FRU information is the physical location of the device on a particular machine. To be able to provide FRU information for a device, the hardware of the machine it is on must be able to “see” it (meaning it must be either a built-in device or in a slot directly attached to the machine).

If the device is a card in an expansion slot, the FRU number is the slot number on the machine. If the device is built-in, the FRU number is the built-in device number, provided one was assigned to it by the manufacturer. Otherwise, the device must be located on an expansion bus.
PCI Macros and Functions

RETURN VALUES

pci_get_fru_info_isc() returns the following values:

PCI_GET_FRU_INFO_BUILT_IN_FRU The device is built-in and was assigned the built-in device number given in fru_info.

PCI_GET_FRU_INFO_BUILT_IN_NO_FRU The device is built-in but was not assigned a built-in device number. fru_info is not valid.

PCI_GET_FRU_INFO_EXPANSION_DEVICE The device is an expansion device located in the slot number given in fru_info.

PCI_GET_FRU_INFO_DEV_NOT_FOUND The device is neither built-in nor found in any expansion slot; it is not in any physical location that the hardware knows about. It must, therefore, be located on a bus that is downstream of a PCI-to-PCI bridge. fru_info is not valid.

PCI_GET_FRU_INFO_NOT_IMPLEMENTED This functionality is not available for the device. fru_info is not valid.

CONSTRAINTS
NAME

pci_get_port_hndl_isc(PCI3) PCI Function – Obtain a system-defined handle for manipulating a range of PCI I/O ports

SYNOPSIS

```c
#include <sys/pci.h>

int pci_get_port_hndl_isc(
    /*IN*/ struct isc_table_type* isc,
    /*IN*/ uint32_t pci_io_addr,
    /*IN*/ uint32_t size,
    /*OUT*/ PCI_PORT_HNDL* phndl);
```

PARAMETERS

- `isc` Pointer to an ISC table associated with the device.
- `pci_io_addr` The address of a range of PCI I/O ports.
- `size` The size of the PCI I/O ports.
- `phndl` A pointer to the location where the routine should place the handle provided by the system.

DESCRIPTION

The `pci_get_port_hndl_isc()` PCI function obtains a system-defined handle for manipulating a range of PCI I/O ports.

The routine can block or sleep and, therefore, should only be called in a thread context.

RETURN VALUES

The routine returns the following values:

- 0 Failure. A handle could not be returned by the system.
- 1 Success. The value pointed to by `phndl` is a valid handle.
CONSTRAINTS

EXAMPLES

The `pci_read_cfg_uint32_isc()` routine reads the card’s configuration space and retrieves the PCI address associated with an I/O port range. This address and size is passed into `pci_get_port_hndl_isc()` to get a handle. The handle is needed to access the port through the `pci_read_port_uint32_isc()` function.

```c
PCI_PORT_HNDL phndl;
uint32_t pci_port_addr;
uint32_t data;

/*
 * get the io port address and mask off unwanted bottom
 * bits
 */
pci_read_cfg_uint32_isc(isc, mydriver_PORT_BASE_REG,
 &pci_port_addr);
pci_port_addr &= ~0x3;

/*
 * get the port handle
 */
if (pci_get_port_hndl_isc
(isc, pci_port_addr, mydriver_PORT_BLOCK_SIZE, &phndl)) {
/*
 * use it for as long as you want,
 * then return it when it is no longer needed
 */
pci_read_port_uint32_isc(isc, phndl, mydriver_PORT_OFFSET,
 &data);
...
pci_unget_port_hndl_isc(isc, pci_port_addr, mydriver_PORT_BLOCK_SIZE, phndl);
}
```

SEE ALSO

`pci_unget_port_hndl_isc(PCI3)`
NAME

pci_read_cfg_uintN_isc(PCI3) PCI Functions - Read unsigned integer from a PCI configuration register

SYNOPSIS

#include <sys/pci.h>

void
pci_read_cfg_uint8_isc (/*IN*/ struct isc_table_type *isc,
/*IN*/ int reg_num,
/*OUT*/ uint8_t *data_read);

void
pci_read_cfg_uint16_isc (/*IN*/ struct isc_table_type *isc,
/*IN*/ int reg_num,
/*OUT*/ uint16_t *data_read);

void
pci_read_cfg_uint32_isc (/*IN*/ struct isc_table_type *isc,
/*IN*/ int reg_num,
/*OUT*/ uint32_t *data_read);

PARAMETERS

isc Pointer to an ISC table associated with the device.
reg_num The offset of a PCI configuration register for the PCI device/function specified by isc. It can be a PCI_CS_* constant, defined in pci.h.
data_read A pointer to an 8-, 16-, or 32-bit location where the routine should place the value.

DESCRIPTION

The pci_read_cfg_uintN_isc PCI functions read an 8-, 16-, or 32-bit unsigned integer from a PCI configuration register for a particular PCI
device/function.

RETURN VALUES

The \texttt{pci_read_cfg_uintN\_isc} routines do not return values.

CONSTRAINTS

EXAMPLE

```c
#include <sys/pci.h>
static void
mydriver_set_io_master (struct isc_table_type * isc)
{
    unsigned short hwid;
    uint16_t old_cmdreg;

    PCI_PORT_HNDL ph;

    pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&old_cmdreg);
    pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND, old_cmdreg |
                              PCI_CMD_IO_SPACE | PCI_CMD_BUS_MASTER);
    ...
}
```

SEE ALSO

\texttt{pci_write_cfg_uintN\_isc}(PCI3)
NAME

pci_read_port_uintN_isc(PCI3) PCI Functions - Read little-endian data from an I/O port

SYNOPSIS

#include <sys/pci.h>
void
pci_read_port_uint8_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*OUT*/ uint8_t *data);

void
pci_read_port_uint16_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*OUT*/ uint16_t *data);

void
pci_read_port_uint32_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*OUT*/ uint32_t *data);

PARAMETERS

isc  Pointer to an ISC table associated with the device.
ph   A port handle previously obtained with a call to
     pci_get_port_hndl_isc().
offset An offset from ph.
data  A pointer to an 8-, 16-, 32-bit location where the
     routine should place the value.
DESCRIPTION

The `pci_read_port_uintN_isc` PCI functions read 8-, 16-, or 32-bit little-endian data for the device/function specified by `isc` from the I/O port represented by the PCI port handle `ph` and offset `offset`. You will probably need to swap bytes if your driver will operate on 16, or 32 bit data.

RETURN VALUES

The `pci_read_port_uintN_isc()` routines do not return values.

CONSTRAINTS

EXAMPLES

```c
#include <sys/pci.h>

#define MY_IOMAP_BASE 0x10
#define MY_PORT_SIZE 0x100
#define MY_IDREG 0x0
#define MY_HWID 0x4850

static void
mydriver_memset(struct isc_table_type *isc)
{
    unsigned short hwid;
    unsigned int port_addr;
    uint16_t old_cmdreg;
    PCI_PORT_HNDL ph;

    isc->mapped = NULL;
    pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&old_cmdreg);
    msg_printf("command reg = 0x%x\n",old_cmdreg);
    pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND, old_cmdreg |
                                PCI_CMD_IO_SPACE | PCI_CMD_BUS_MASTER);
    pci_read_cfg_uint32_isc(isc,MY_IOMAP_BASE,&port_addr);
    port_addr &= ~3;
    if (pci_get_port_hndl_isc(isc, port_addr,
                              MY_PORT_SIZE, &ph)) { 
        pci_read_port_uint16_isc(isc,ph,MY_IDREG,&hwid); 
        if (!(hwid & MY_HWID) != MY_HWID) { 
            return -1;
        }
    }
```
isc->mapped=(int)ph.hndl;
} else {
    msg_printf("pci_get_port_hndl_isc() failed\n");
    return -1;
}
return 0;

SEE ALSO

pci_write_port_uintN_isc(PCI3)
NAME

pci_unget_port_hndl_isc(PCI3) PCI Function – Delete a system-defined handle for manipulating a range of PCI I/O ports

SYNOPSIS

```c
#include <sys/pci.h>

int
pci_unget_port_hndl_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint32_t    pci_io_addr,
    /*IN*/ uint32_t    size,
    /*IN*/ PCI_PORT_HNDL phndl);
```

PARAMETERS

- `isc`: Pointer to an ISC table associated with the device.
- `pci_io_addr`: The address of a range of PCI I/O ports.
- `size`: The size of the PCI I/O ports.
- `phndl`: A handle obtained for these parameters by a previous call to `pci_get_port_hndl_isc()`.

DESCRIPTION

The `pci_unget_port_hndl_isc()` PCI function deletes a system-defined handle for manipulating a range of PCI I/O ports.

RETURN VALUES

The routine returns the following values:

- `0`: Failure. The handle could not be deleted by the system.
- `1`: Success. The handle was deleted by the system.

CONSTRAINTS
EXAMPLES

See pci_get_port_hndl_isc(PCI3).

SEE ALSO

pci_get_port_hndl_isc(PCI3)
NAME

pci_write_cfg_uintN_isc(PCI3) PCI Functions - Write unsigned integer to a PCI configuration register

SYNOPSIS

```c
#include <sys/pci.h>

void
pci_write_cfg_uint8_isc(
    /*IN*/ struct isc_table_type* isc,
    /*IN*/ int  reg_num,
    /*IN*/ uint8_t data_write);

void
pci_write_cfg_uint16_isc(
    /*IN*/ struct isc_table_type* isc,
    /*IN*/ int  reg_num,
    /*IN*/ uint16_t data_write);

void
pci_write_cfg_uint32_isc(
    /*IN*/ struct isc_table_type* isc,
    /*IN*/ int  reg_num,
    /*IN*/ uint32_t data_write);
```

PARAMETERS

isc Pointer to an ISC table associated with the device.

reg_num The number of a PCI configuration register for the PCI device/function specified by isc. It can be a PCI_CS_* constant, defined in pci.h.

data_write The 8-, 16-, or 32-bit value to be written.

DESCRIPTION

The `pci_write_cfg_uintN_isc()` PCI functions write an 8-, 16-, or 32-bit unsigned integer to a PCI configuration register for a particular PCI device or function.
PCI Reference Pages
PCI Macros and Functions

RETURN VALUES

The `pci_write_cfg_uintN_isc()` routines do not return values.

CONSTRAINTS

SEE ALSO

`pci_read_cfg_uintN_isc(PCl3)`
NAME

pci_write_port_uintN_isc(PCI3) PCI Functions – Write little-endian data to an I/O port

SYNOPSIS

#include <sys/pci.h>

void
pci_write_port_uint8_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*IN*/ uint8_t data);

void
pci_write_port_uint16_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*IN*/ uint16_t data);

void
pci_write_port_uint32_isc(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*IN*/ uint32_t data);

PARAMETERS

isc Pointer to an ISC table associated with the device.

ph A port handle previously obtained with a call to pci_get_port_hndl_isc().

offset An offset from ph.

data The 8-, 16-, or 32-bit value to be written.
DESCRIPTION

The `pci_write_port_uintN_isc()` PCI functions write 8-, 16-, or 32-bit little-endian data for the device/function specified by `isc` to the I/O port represented by the PCI port handle `ph` and offset `offset`. You will probably need to swap bytes if your driver is operating on 16- or 32-bit data.

RETURN VALUES

The `pci_write_port_uintN_isc()` routines do not return values.

CONSTRAINTS

SEE ALSO

`pci_read_port_uintN_isc(PCI3)`
NAME

READ_REG_UINTn_ISC(PCI3) PCI Functions - Read and byte-swap data from a little-endian bus

SYNOPSIS

```c
#include <sys/pci.h>
void
READ_REG_UINT8_ISC (  
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint8_t * addr,
    /*OUT*/ uint8_t * data);

void
READ_REG_UINT16_ISC (  
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint16_t * addr,
    /*OUT*/ uint16_t * data);

void
READ_REG_UINT32_ISC (  
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint32_t * addr,
    /*OUT*/ uint32_t * data);
```

PARAMETERS

- **isc**: Pointer to an ISC table associated with the driver.
- **addr**: A pointer to the address of the bus data. It must be one of:
  - A virtual address mapped with `map_mem_to_host()`.
  - A mapped offset in the automatically mapped first-base-address register range contained in `isc->if_reg_ptr`.
    - Only the first nonzero 32-bit-wide memory base register found can be mapped, starting in the range 0x10 and searching up through 0x24 (the six possible base address register locations in...
configuration space).
— If that base-register's size is in excess of 8 KB, it is not mapped and isc->if_reg_ptr is set to NULL. In this case, the driver must map the base register it wants to use.

• The address of a memory buffer shared between the driver and a little-endian bus master.

\textit{data} A pointer to an 8-, 16-, or 32-bit location where the routine should place the resultant data.

**DESCRIPTION**

The \texttt{READ_REG_UINTn_ISC()} PCI functions are macros that read and byte-swap data located at \textit{addr} from a little-endian bus and place it in \textit{data}.

If the PCI adapter that your card is running under has directly mapped the PCI memory space into driver-accessible system I/O space, you can improve the performance of \texttt{READ_REG_UINTn_ISC()} if you define the flag \texttt{PCI_LITTLE_ENDIAN_ONLY} prior to including the \texttt{pci.h} header file. This causes \texttt{READ_REG_UINTn_ISC()} to perform a simple byte swap instead of calling a function that tests byte ordering.

**RETURN VALUES**

The \texttt{READ_REG_UINTn_ISC()} routines do not return values.

**CONSTRAINTS**

**EXAMPLES**

```c
#include <sys/pci.h>
#define MY_REGISTER_OFFSET 0x40
/* the address of some register on my card */

uint8_t data8;
uint8_t *addr = isc->if_reg_ptr + MY_REGISTER_OFFSET;
/* virtual address plus an offset */
```
/*
 * code accessing registers is expanded inline
 */
READ_REG_UINT8_ISC(isc, addr, &data8);

SEE ALSO

WRITE_REG_UINTn_ISC(PCI3)
NAME

WRITE_REG_UINTn_ISC(PCI3) PCI Functions – Byte-swap and write data to a little-endian bus

SYNOPSIS

```c
#include <sys/pci.h>

void
WRITE_REG_UINT8_ISC(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint8_t *addr,
    /*IN*/ uint8_t data);

void
WRITE_REG_UINT16_ISC(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint16_t *addr,
    /*IN*/ uint16_t data);

void
WRITE_REG_UINT32_ISC(
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ uint32_t *addr,
    /*IN*/ uint32_t data);
```

PARAMETERS

isc
   Pointer to an ISC table associated with the device.

addr
   A pointer to the output address. It must be one of the following:
   
   • A virtual address mapped with map_mem_to_host()
   
   • A mapped offset in the automatically mapped first-base-address register range contained in isc->if_reg_ptr
     
     ‒ Only the first nonzero 32-bit-wide memory base register found can be mapped, starting in the
PCI Reference Pages
PCI Macros and Functions

range 0x10 and searching up through 0x24 (the six possible base address register locations in configuration space).

— If that base-register's size is in excess of 8 KB, it is not mapped and isc->if_reg_ptr is set to NULL. In this case, the driver must map the base register it wants to use.

• The address of a memory buffer shared between the driver and a little-endian bus master.

data | The 8-, 16-, or 32-bit data to be written.

DESCRIPTION

The WRITE_REG_UINTn_ISC() PCI functions are macros that byte-swap and write data to a little-endian bus or to a host memory area shared by the driver and a little-endian bus master, located at addr.

If the PCI adapter that your card is running under has directly mapped the PCI memory space into driver-accessible system I/O space, you can improve the performance of WRITE_REG_UINTn_ISC() if you define the flag PCI_LITTLE_ENDIAN_ONLY prior to including the pci.h header file. This causes WRITE_REG_UINTn_ISC() to perform a simple byte swap instead of calling a function that tests byte ordering.

RETURN VALUES

The WRITE_REG_UINTn_ISC() routines do not return values.

CONSTRAINTS

EXAMPLES

#include <sys/pci.h>
#define MY_REGISTER_OFFSET 0x40

uint8_t data8;
uint8_t *addr = isc->if_reg_ptr + MY_REGISTER_OFFSET;
/* virtual address plus an offset */
/* 
 * code accessing registers is expanded inline
 */
WRITE_REG_UINT8_ISC(isc, addr, &data8);

SEE ALSO

READ_REG_UINTn_ISC(PCI3)
NAME

PCI_ERRATA-1(PCI5) –

Memory Coherency Issues

Certain combinations of WSIO mapping service calls can interact with PCI masters on C class and J class processors to create an inconsistent view of memory.

It is possible for prefetching of host memory by the PA hardware chipsets to result in a PCI master reading stale data, even though the proper dma_sync() calls have been made. The problem does NOT occur if:

1. The PCI master does normal Memory Read transactions, i.e., the master does not master Memory Read Multiple (MRM) or Memory Read Line (MRL) transactions.
2. The mapping is done with wsio_map() with flags IO_NO_SEQ and IO_SAFE set, regardless of the type of transactions the PCI master uses.
3. The mapping is done with wsio_fastmap() and the PCI master does NOT use MRM or MRL transactions.

DETAILS

There are two hardware prefetch buffers in the PA hardware chipset between memory and any PCI device. One is in the system’s PCI bridge chip, and the other is system’s GSC I/O bridge chip (which connects to the PA side of the PCI bridge chip). The I/O bridge chip has a cache line size prefetch buffer for each I/O (GSC) slot.

For the following discussion assume that an I/O TLB was mapped using either wsio_fastmap() or wsio_map without IO_NO_SEQ and IO_SAFE flag bits set.

When a PCI bus master runs an MRM or MRL transaction, the following events happen:

1. The PCI bridge chip requests a cache line (8 words), starting at the PCI master requested start address, from the I/O bridge chip. Since the PCI master is running an MRM or MRL transaction, this request is made with a prefetch hint enabled for the I/O bridge chip. In addition it requests subsequent cache lines from the I/O bridge chip,
with the exact number of extra lines dependent upon whether an MRM or MRL PCI transaction is in progress, and whether or not the end of a physical page is near (the PCI bridge chip will not prefetch past the end of a page).

2. The I/O bridge chip, for each cache line requested, fills the request immediately from it's own prefetch buffer if the requested line resides there, or gets the cache line from processor memory. It then immediately prefetches the next line from processor memory into its prefetch buffer.

Two problems exist. The first case is when the PCI bridge chip has requested the cache line at the end of a physical page (note that this does not imply that the PCI device, itself, has requested the cache line at the end of the page), e.g., 0x0Fe0. The PCI bridge chip, incorrectly requests this cache line with the prefetch hint enabled. The I/O bridge chip, to avoid fetching onto a possible non-existent page, but needing to do something with the prefetch hint enabled, prefetches the first cache line of the page, e.g., 0x0000. In the example case, stale data can be read if the next request from the PCI master is for address 0x0000, which has just been incorrectly prefetched.

This case is fairly easy to hit. A driver might have control information consisting of a list of multiple structures that just fill a physical page. If the PCI bridge reads the last cache line of the page, followed by the driver re-writing the list, doing a dma_sync(), and then directing the PCI master to re-read the list, the PCI master will read stale data in the first cache line.

The second case is where a driver has two adjacent data structures on the same page. The PCI master reads from the first data structure. The PCI bridge chip and the I/O bridge chip have prefetched such that the I/O bridge chip has a cache line in it's prefetch buffer that actually resides in the second data structure. If the PCI master then reads that particular address, it may have stale data (depending upon the sequence the driver follows in updating it vis-a-vis the PCI master's access).

In both cases, if the mapping is done using wsio_map() with IO_NO_SEQ and IO_SAFE flag bits set, no problem exists (because the I/O bridge chip ignores the prefetch hint when the IO TLB is set up by WSIO mapping services with this mapping).

Note that the IO_NO_SEQ and IO_SAFE flag bits will degrade MRM and MRL performance by about a factor of two for the page(s) in this type of mapping. If it is absolutely necessary, for performance reasons, to use...
wsio_fastmap() or wsio_map() without the IO_NO_SEQ and IO_SAFE flag bits set the coherency problem can be prevented by having the PCI master read a different address which will reset the I/O bridge chip’s prefetch buffer. A read of any address using MRM, MRL or a normal read transaction by the PCI master will accomplish this, so you might have the PCI master re-read the previous cache line and then throw it away.

SEE ALSO

wsio_map(WSIO3), wsio_fastmap(WSIO3), pci_errata-2
NAME

PCI_ERRATA-2 (PCI5) –

PCI Transaction Ordering

Due to interaction between the host bus, PCI bridge chips, and the PCI bus, in certain situations, the Producer Consumer model requirements defined in the PCI 2.1 Specification may not be met. For more detailed information refer to the discussion on Transaction Ordering in the PCI Chapter of the HP-UX Driver Development Guide.

SEE ALSO

pci_errata-1 (PCI5)
NAME

PCI_ERRATA-3 (PCI5) -

PCI Configuration Cycle Retry Problem

The system's PCI bridge chip holds IRDY too long on config write retry. This problem has only been seen in simulation with some revisions of the PCI bridge chip used on the B1000, C3000, J5000, and the N-Class servers.

This problem occurs when:

1. A device retries a configuration cycle, and if
2. the device asserts DEVSEL and STOP during the same cycle, and if
3. that cycle is not the one immediately following the address cycle then the PCI bridge chip will ignore the RETRY, believe that the card never asserted DEVSEL response, which will cause a master abort.

If the card asserts DEVSEL for one or more cycles before it asserts STOP, the problem does not occur.

In PCI bridge chips exhibiting this behavior, the most likely result is an HPMC or panic.
7 SCSI Reference Pages
SCSI Services is a set of commonly used SCSI functions that allow device and interface drivers to be much smaller and more supportable. In addition to providing most commonly used SCSI functions, WSIO SCSI Services also provides a supported pass-through mechanism.
NAME

dd_close(SCSI_DRV) SCSI Function - Driver-specific processing during close

SYNOPSIS

void dd_close (dev_t dev);

PARAMETERS

dev The device number.

DESCRIPTION

The dd_close() SCSI function is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_close field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_close() does not return a value.

CONSTRAINTS

SEE ALSO

scsi_lun_close(SCSI3), scsi_ddsw(SCSI4)
NAME

dd_done(SCSI_DRV) SCSI Function – Driver-specific post-I/O processing

SYNOPSIS

int dd_done (struct buf *bp);

PARAMETERS

bp buf structure.

DESCRIPTION

The dd_done() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_done field of the scsi_ddsw structure. See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_done() is declared as returning int; however, its return is not used by SCSI services.

SEE ALSO

biodone(KER2), scsi_action(SCSI3), scsi_ddsw(SCSI4)
NAME

dd_ioctl(SCSI_DRV)  SCSI Function - Driver-specific ioctl processing

SYNOPSIS

int dd_ioctl (dev_t dev, int cmd, caddr_t data, int flags);

PARAMETERS

dev  Device number of the associated device
cmd  Driver command to execute
data  Pointer to the command arguments
flags  The file-access flags.

DESCRIPTION

The dd_ioctl() routine is provided by the driver writer. It can have any
unique name. You pass the name to SCSI Services by specifying it in the
dd_ioctl field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_ioctl() is expected to return the following values:

0  Successful completion.
<>0  Error. The value is expected to be an errno value.

CONSTRAINTS

SEE ALSO

scsi_cmd(SCSI3), scsi_init_inquiry_data(SCSI3),
scsi_ioctl(SCSI3)
NAME

dd_ioctl_okay(SCSI_DRV) SCSI Function – Disallow ioctl commands through the pass-through driver

SYNOPSIS

int dd_ioctl_okay (dev_t dev, int cmd, caddr_t data, int flags);

PARAMETERS

dev Device number of the associate device

cmd Driver command to execute

data Pointer to command parameter

flags The file-access flags

DESCRIPTION

The dd_ioctl_okay() SCSI function is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_ioctl_okay field of the scsi_ddsw structure. See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_ioctl_okay() is expected to return the following values:

PT_OKAY Successful completion.

0 Error.

CONSTRAINTS

SEE ALSO

scsi_ioctl(SCSI3)
NAME

dd_open(SCSI_DRV) SCSI Function - Driver-specific open processing

SYNOPSIS

dd_open(dev, oflags);

PARAMETERS

dev The device number of the device to be opened

oflags The flags passed in the open call

DESCRIPTION

The dd_open() SCSI function is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_open field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_open() is expected to return the following values:

0 Successful completion.

<>0 Error. The value is expected to be an errno value.

CONSTRAINTS

SEE ALSO

m_scsi_lun(SCSI3), major(KER2), scsi_cmdx(SCSI3),
scsi_init_inquiry_data(SCSI3), scsi_lun_open(SCSI3)
NAME

dd_pass_thru_done (SCSI_DRV)  SCSI Function – Driver-specific notation of pass-through I/O

SYNOPSIS

int dd_pass_thru_done (struct buf * bp);

PARAMETERS

bp         buf structure.

DESCRIPTION

The dd_pass_thru_done() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_pass_thru_done field of the scsi_ddsw structure. See HP-UX Driver Development Guide for details;

CONSTRAINTS

RETURN VALUES

dd_pass_thru_done() is declared as returning int; however, the return value is not used by SCSI services.
NAME

dd_pass_thru_okay (SCSI_DRV) SCSI Function - Driver-specific control of pass-through I/O

SYNOPSIS

dd_pass_thru_okay (dev_t dev, struct sctl_io * sctl_io);

PARAMETERS

dev The device number
sctl_io Struct containing ioctl information

DESCRIPTION

The dd_pass_thru_okay() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_pass_thru_okay field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_pass_thru_okay() is expected to return the following values:

PT_OKAY Successful completion.
0 Error.

CONSTRAINTS
NAME

dd_read(SCSI_DRV)  SCSI Function  - Driver-specific read function

SYNOPSIS

int dd_read (dev_t dev, struct uio * uio);

PARAMETERS

dev        The device number
uio        Struct containing transfer information

DESCRIPTION

The dd_read() routine is provided by the driver writer. It can have any
unique name. You pass the name to SCSI Services by specifying it in the
dd_read field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_read() is expected to return the following values:

0        Successful completion.
<>0       Error. The value is expected to be an errno value.

CONSTRAINTS

SEE ALSO

scsi_read(SCSI3)
NAME

dd_start(SCSI_DRV) SCSI Function - Driver-specific start routine

SYNOPSIS

struct buf * (d_start) dd_start (struct scsi_lun *lp,
    struct scb *scb);

PARAMETERS

lp The open LUN structure
scb Struct containing state information for I/O

DESCRIPTION

The dd_start() routine is provided by the driver writer. It can have any
unique name. You pass the name to SCSI Services by specifying it in the
dd_start field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_start() is expected to return the following values:
struct buf *bp
    Successful completion.
NULL Error.

CONSTRAINTS
NAME

dd_strategy(SCSI_DRV)  SCSI Function  - Driver-specific I/O routine

SYNOPSIS

int dd_strategy (struct buf *bp struct scsi_lun *lp);

PARAMETERS

struct buf *bp  transfer buf header

DESCRIPTION

The dd_strategy() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_strategy field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_strategy() is expected to return the following values:

  0     Successful completion.

-1     Error.

WARNINGS

dd_strategy() must exist (be defined as non-NULL in the scsi_ddsw structure) if your driver calls scsi_strategy(). scsi_strategy() calls dd_strategy while holding lun_lock.

SEE ALSO

physio(KER2) dd_read(SCSI_DRV), dd_write(SCSI_DRV), scsi_enqueue(SCSI3), scsi_strategy(SCSI3)
NAME

dd_write(SCSI3) SCSI Function - dd_write - driver-specific write routine

SYNOPSIS

int dd_write (dev_t dev, struct uio * uio);

PARAMETERS

dev The device number
uio Structure containing transfer information

DESCRIPTION

The dd_write() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_write field of the scsi_ddsw structure.

See HP-UX Driver Development Guide for details;

RETURN VALUES

dd_write() is expected to return the following values:

0 Successful completion.
<>0 Error. The value is expected to be an errno value.

SEE ALSO

physio(KER2), scsi_write(SCSI3)
NAME

m_scsi_lun(SCSI13) SCSI Function - Get scsi_lun pointer

SYNOPSIS

m_scsi_lun (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

m_scsi_lun() returns the scsi_lun pointer corresponding to dev.

RETURN VALUES

m_scsi_lun() returns the following values:

0 Error.

<>0 Pointer to the SCSI logical unit associated with dev.
NAME

`scsi_action(SCRI)` SCSI Function - Give I/O completion information to SCSI Services

SYNOPSIS

```c
#include <sys/scsi_ctl.h>

scsi_action (struct buf *bp, int flags, int error, int msecs);
```

PARAMETERS

- **bp**
  - Transfer buf header

- **flags**
  - The following bit values can be combined for `flags`:
    - `SA_ANY` - Wild card entry for matching parameters.
    - `SA_DISABLE_TAGS` - Initiate the transition to non-tagged operation for the device. This is used to recover from tagged queuing problems.
    - `SA_DONE` - Call `dd_done()` and `biodone()`.
    - `SA_IGNORE_MAX_RETRIES` - Retry I/O independently of `scb->max_retries`. This is used when a command fails for a reason unrelated to the command, such as unit attention, power-on, or reset.
    - `SA_LOG_IT_ALWAYS` - Always log an I/O attempt record to `dmesg`.
    - `SA_LOG_IT_NEVER` - Never log an I/O attempt record to `dmesg`.
    - `SA_LOG_IT_SOMETIMES` - Log an I/O attempt record to `dmesg` if `SCB_DONT_PRINT` is true.
    - `SA_NONE` - Value used for undefined fields.
SA_PANIC        Execute panic(error).
SA_REINIT       Go to reinitialization state.
SA_RETRY        Retry the I/O if scb->max_retries
                 has not been exceeded.

The default is SA_DONE + SA_LOG_IT_NEVER.

error          errno value.
msecs

DESCRIPTION

scsi_action() must ultimately be called after all I/O attempt
completions (as in a retry situation). It exists only because there is too
much information needed by SCSI Services from the device driver's
action routine to encode easily in a single integer return value. The
arguments determine: whether or not the I/O attempt record is logged to
the dmesg buffer, whether tags should be disabled, and whether to retry
the I/O, consider it to be completed, or panic. It is either entered directly
into the device driver's status action list or called at the end of the
function that is in the status action list.

scsi_action() is called by device drivers and SCSI services internally
in both interrupt and process contexts.

The scsi_action() function appears not to have any real protection
issues itself; it mainly operates on the request structures (buf and scb).
The dd_done() function is called from scsi_action() and some of the
device driver dd_done() functions do need the protection.

The constants and data structures used are specified in the header file,
../wsio/scsi_ctl.h.

Logging, as a result of SA_LOG_IT_ALWAYS or SA_LOG_IT_SOMETIMES sent
to scsi_status(), causes scsi_log_io() to be invoked. This routine
records the I/O attempt and its results in the dmesg buffer. Output is
controlled by scsi_log_mask and scsi_log_nbytes. An attempt is
made to conserve dmesg buffer space by outputting only differences
between successive retries of the same I/O.

RETURN VALUES

scsi_action() returns the following values:
msecs if (flags & SA_RETRY)
-1 Normal Completion
panic() if (flags & SA_PANIC)

SEE ALSO

biodone(KER2), panic(KER2)
NAME

scsi_cmd(SCSI3) SCSI Function - Prepare driver-generated I/O requests

SYNOPSIS

#include <sys/scsi_ctl.h>

scsi_cmdx (dev_t dev, ubit32 flags, int cdb_len,
           ubit8 * cdb, int nbytes, void * addr,
           ubit32 msecs ubit32 retries, int * pErr);

PARAMETERS

dev Device used to find correct LUN and target.
flags Read, 6-, 10-, or 12-byte cdb, or action.
cdb_len Length of the cdb 6,10,12.
cdb SCSI command data block.
nbytes If zero, there is no data phase.
addr Buffer for read data return.
retries Number of retries.
pErr If not NULL, then contains the error returned by the operation (in bp->b_error).

DESCRIPTION

The scsi_cmd() SCSI function is used for driver-generated I/O requests. It is a wrapper for scsi_cmdx() which it calls setting the two additional parameters to NULL and 0.

Used by device drivers and SCSI services internally, this function must be called in the process context and may block. The function is not called from within any critical section.

Refer to “NAME” for details.
RETURN VALUES

scsi_cmd() returns the following values:

N Number of bytes transferred.
-1 Error.

SEE ALSO

biowait(KER2), scsi_ctl(7), scsi_init_inquiry_data(SCSI3),
scsi_cmdx(SCSI3), scsi_strategy(SCSI3)
NAME

scsi_cmdx(SCSI3) SCSI Function – Prepare driver-generated I/O requests

SYNOPSIS

#include <sys/scsi_ctl.h>

scsi_cmdx (dev_t dev, int flags, int cdb_len, u_char * cdb, 
    int nbytes, void * addr, u_int msecs, 
    u_int retries, int * Err, 
    struct status_action *sa, int n);

PARAMETERS

addr Buffer for read data return.

cdb SCSI command data block.

cdb_len Length of the cdb 6,10,12.

dev Device used to find correct LUN and target.

Err If not NULL, then contains the error returned by the 
    operation (in bp->b_error)

flags Read, 6-, 10-, or 12-byte cdb, or action.

msecs Assigned to sclt_ioc->max_msecs. Zero means no 
    timeout.

n Status count.

nbytes If zero, there is no data phase.

sa If NULL, there is no action to match or take.

retries Number of retries.

DESCRIPTION

The scsi_cmdx() SCSI function is used for driver-generated I/O 
requests. It creates and builds a sclt_ioc and a bp, attaches the sclt_ioc 
to the bp, forwards the bp to the scsi_strategy() routine, and cleans 
up when the I/O is completed.
The `scsi_cmdx()` routine is used by drivers to perform initialization or ioctl types of operations. It is also used within SCSI Services to perform `scsi_init_inquiry_data()`, `scsi_mode_sense()`, and `scsi_mode_select()`.

Used by device drivers and SCSI services internally, this function must be called in the process context and may block. The function is not called from within any critical section.

`scsi_cmdx()` allocates a `bp` structure and a `sctl_io` structure. It sets `B_SCSI_CMD` in `bp->b_flags` and places a pointer to the `sctl_io` structure into `bp->b_offset`. For a detailed discussion, see the `sctl_io` portion of the SCSI pass-through driver in `scsi_ctl(7)`.

The parameter `max_msecs` is assigned to `sctl_io->max_msecs`, which itself is assigned to `scb->msecs`. Similarly, `max_retries` is assigned to `sctl_io->max_retries`, which itself is assigned to `scb->max_retries`.

To perform the I/O, `scsi_cmdx()` calls `scsi_strategy()`, then `scsi_iowait()`. Upon completion, it releases the `bp` and `sctl_io` structures, prior to returning to the caller.

**RETURN VALUES**

`scsi_cmdx()` returns the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of bytes transferred.</td>
</tr>
<tr>
<td>-1</td>
<td>Error.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`biowait(KER2)`, `scsi_ctl(7)`, `scsi_init_inquiry_data(SCSI3)`, `scsi_strategy(SCSI3)`
NAME

`scsi_ddsw(SCSI4)` SCSI Structure - SCSI device-switch structure

SYNOPSIS

```c
#include <sys/scsi_ctl.h>

struct scsi_ddsw
{
    u_char blk_major;
    u_char raw_major;
    int dd_lun_size;
    int (*dd_open)();
    void (*dd_close)();
    int (*dd_strategy)();
    int (*dd_read)();
    int (*dd_write)();
    int (*dd_ioctl)();
    struct buf (*dd_start)();
    int (*dd_done)();
    int (*dd_pass_thru_okay)();
    int (*dd_pass_thru_done)();
    int (*dd_ioctl_okay)();
    struct status_action *dd_status_list;
    int dd_status_cnt;
    ubit32 (*dd_flags;
    wsio_drv_info_t *wsio_drv;
};
```

PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blk_major</td>
<td>Obsolete field, not initialized.</td>
</tr>
<tr>
<td>raw_major</td>
<td>Obsolete field, not initialized.</td>
</tr>
<tr>
<td>dd_lun_size</td>
<td>The number of bytes to be allocated and attached to the open device tree when driver_open() is first executed.</td>
</tr>
<tr>
<td>dd_open()</td>
<td>Pointer to driver supplied routine.</td>
</tr>
<tr>
<td>dd_close()</td>
<td>Pointer to driver supplied routine.</td>
</tr>
<tr>
<td>dd_strategy()</td>
<td>Pointer to driver supplied routine.</td>
</tr>
</tbody>
</table>
DESCRIPTION

In order to use SCSI Services effectively, a SCSI driver must define its `scsi_ddsw` device-switch structure. This structure contains pointers to special `dd` routines, some of which are executed indirectly by the standard driver routines, such as `driver_read`. The structure is passed to SCSI Services routines from the `driver_open` routine, which calls the `scsi_lun_open()` SCSI Services routine.

SCSI Services has been set up to control the housekeeping and other processing in the SCSI interface. Therefore, you should have the standard driver routines restrict their operation to calling the appropriate SCSI Services routine. Special processing and customization should all be handled in the special `dd` routines.

EXAMPLE

Here is an example of an initialized declaration of the `scsi_ddsw`:

The first example is the declaration of your driver’s version of the `dd` routines that can be called by SCSI Services. The routine names are arbitrary. The names in comments are the field names of the `scsi_ddsw` structure.

```
int mydriver_dd_open();    /* dd_open */
```
void mydriver_dd_close();  /* dd_close */
int mydriver_dd_strategy(); /* dd_strategy */
int mydriver_dd_read();   /* dd_read */
int mydriver_dd_write();  /* dd_write */
int mydriver_dd_ioctl();  /* dd_ioctl */
struct buf mydriver_dd_start(); /* dd_start */
int mydriver_dd_done();   /* dd_done */
int mydriver_dd_pass_thru_okay(); /* dd_pass_thru_okay */
int mydriver_dd_pass_thru_done(); /* dd_pass_thru_done */
int mydriver_dd_ioctl_okay();/* dd_ioctl_okay */

The following example shows the scsi_ddsw structure. Specify NULL for routines that are not defined (that is, that you are not providing). The first two fields specify the block and character major numbers; they are filled in by the call in driver_dev_init() to the SCSI Services routine scsi_ddsw_init(). The last field points to the wsio_drv_info_t structure. The first name in each comment is the field name of the scsi_ddsw structure element.

struct scsi_ddsw mydriver_ddsw =
{
    NODEV,    /* blk_major - mydriver_dev_init sets */
    NODEV,    /* raw_major - mydriver_dev_init sets */
    sizeof(struct mydriver_lun),   /* dd_lun_size */
    mydriver_dd_open,   /* dd_open */
    mydriver_dd_close,  /* dd_close */
    mydriver_dd_strategy,/* dd_strategy */
    NULL,  /* dd_read */
    NULL,  /* dd_write */
    mydriver_dd_ioctl,  /* dd_ioctl */
    mydriver_dd_start,  /* dd_start */
    mydriver_dd_done,   /* dd_done */
    mydriver_dd_pass_thru_okay, /* dd_pass_thru_okay */
    mydriver_dd_pass_thru_done, /* dd_pass_thru_done */
    mydriver_dd_ioctl_okay, /* dd_ioctl_okay */
    mydriver_dd_status_list, /* dd_status_list */
    sizeof(mydriver_dd_status_list)/sizeof(mydriver_dd_status_list[0]),   /* dd_status_cnt */
    mydriver_dd_flags,  /* dd_flag bits DD_DDG */
    &mydriver_wsio_info
    /* For Diagnostics Logging; NULL means errors print in
    * dmesg */
};
SEE ALSO

scsi_lun_open(SCSI3)
NAME

scsi_dequeue(SCSI3) SCSI Function – Remove I/O requests from queues maintained by SCSI Services

SYNOPSIS

struct buf * scsi_dequeue (struct buf **qp, int where);

PARAMETERS

qp Pointer to the head of a circular list.
where Location to extract from.

RETURN VALUES

scsi_dequeue() returns the following values:
<>NULL Successful completion. The value is the removed struct address.
NULL Failure.

EXAMPLE

struct buf *
sf_start(lp, scb)
struct scsi_lun *lp;
struct scb *scb;
{
    struct sf_lun *llp = lp->dd_lun;
    struct buf *bp;

    /* Protect scb_q */
    scsi_lun_lock(lp);
    bp = scsi_dequeue(&lp->scb_q, HEAD);
    scsi_lun_unlock(lp);

    if (bp == NULL)
        return NULL;

    if (bp->b_flags & B_READ)
    {
        scb->cdb[0] = CMDread;
    }
} else {
    scb->cdb[0] = CMDwrite;
}
...
return bp;
}
NAME

scsi_dequeue_bp(SCSI3) SCSI Function - Dequeue a specific element from a circular list

SYNOPSIS

scsi_dequeue_bp (struct buf ** qp, struct buf * bp);

PARAMETERS

qp Pointer to circular list.
bp Specific buf to remove from the list.

DESCRIPTION

The scsi_dequeue_bp() is externally available to dequeue a particular bp from a circular list. It is intended for use with LVM’s B_PFTIMEOUT.

This is primarily for the scsi_c700 interface driver to “abort” an I/O from its select_q.

scsi_dequeue_bp() may be called in either the process or interrupt context. The function must be called within a critical section.

RETURN VALUES

scsi_dequeue_bp() returns the following values:

<>NULL Successful completion. The value is the removed struct address.
NULL Failure.
NAME

scsi_enqueue(SCSI3)  SCSI Function  – Add buffer bp to a circular list

SYNOPSIS

#include <sys/scsi_ctl.h>

void scsi_enqueue (struct buf **qp, struct buf *bp, int where);

PARAMETERS

qp  A pointer to the list header which is a pointer to the
    head of the list. The tail of the list is (*qp)->av_back.

bp  The struct to be added to the list

where  If where equals HEAD, bp is added to the tail of the list;
        otherwise, it is inserted at the head of the list.

DESCRIPTION

The scsi_enqueue() routine is used to queue a buf buffer bp (and an
scb which has its link in the same location as the bp) to a number of
linked lists maintained by SCSI Services: nexus_q, scb free lists,
retry_list, tag_q, busp->select_q. Note that scsi_disk uses
disksort_enqueue() rather than this routine for its list of pending bps.

scsi_enqueue() may be called in either the process or interrupt context.
It must be called within a critical section. For scsi_lun list
manipulation, scsi_lun_lock() and scsi_lun_unlock() must be
called around this function.

RETURN VALUES

scsi_enqueue() is a void function.

SEE ALSO

scsi_dequeue(SCSI3)
NAME

scsi_init_inquiry_data(SCSI3) SCSI Function - Perform the first Inquiry request on a device

SYNOPSIS

scsi_init_inquiry_data();

PARAMETERS

None.

DESCRIPTION

The scsi_init_inquiry_data() SCSI routine is called by a device driver from its dd_open() routine to perform the first SCSI Inquiry request on the device. It returns the SCSI Inquiry data from the device to the lp->inquiry_data buffer. It may return an error. However, success does not imply that there is no more pending sense data. In fact, the SCSI-2 standard encourages devices not to give Check Condition status on Inquiry, but to defer it until a subsequent command. Also, if the inquiry data had already been cached as a result of a pass-through driver open or SIOC_INQUIRY, this may not even result in I/O.

Used by device drivers, this function must be called in the process context and may block. The function is not called from within any critical section. It verifies that no spinlocks are held with SD_ASSERT().

It uses lun lock to protect lp->state while testing for L_INIT_INQUIRY. It calls scsi_sleep() until this state flag is cleared, at which time it sets the flag. When the inquiry is completed, it clears the flag and calls wakeup().

RETURN VALUES

scsi_init_inquiry_data() returns the following values:

0 Successful completion.
<>0 Error.
NAME

csci_ioctl(SCSI3) SCSI Function - Standard SCSI ioctl routine

SYNOPSIS

#include "../h/scsi.h"

csci_ioctl (dev_t dev, int cmd, caddr_t data, [, int flags]);

PARAMETERS

cmd The ioctl command. It can be one of the commands listed in DESCRIPTION or it can be one that is supported by the driver's dd_ioctl() routine.

data Pointer to the command argument

dev Device number of the associate device

flags The file access flags

DESCRIPTION

The csci_ioctl() SCSI routine simplifies the job of the device driver. Iocls which are supported by all device drivers are implemented here to insure consistency from one driver to the next, and to minimize maintenance costs.

Note that other ioctl commands may be supported by a particular driver's dd_ioctl() routine, which is invoked if the command is one that csci_ioctl() does not recognize.

Used by device drivers, this function must be called in the process context and may block. The function is not called from within any critical section.

RETURN VALUES

scsi_ioctl() returns the following values:

- 0 Successful completion.
- -1 Error.

scsi_ioctl() supports the following ioctl commands (defined in the
SCSI_Inquiry

The SCSI standard inquiry information for the device is copied to the passed inquiry data structure. The structures inquiry, inquiry_2, and inquiry_data are defined in the ../h/scsi.h header file.

SIOC_EXCLUSIVE

Gain/release exclusive access mode.

The int parameter is an integer that may contain one of the following values:

<table>
<thead>
<tr>
<th>int</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Release exclusive access to logical unit</td>
</tr>
<tr>
<td>1</td>
<td>Gain exclusive access to logical unit</td>
</tr>
<tr>
<td>2</td>
<td>Release exclusive access to target</td>
</tr>
<tr>
<td>3</td>
<td>Gain exclusive access to target</td>
</tr>
<tr>
<td>4</td>
<td>Release exclusive access to bus</td>
</tr>
<tr>
<td>5</td>
<td>Gain exclusive access to bus</td>
</tr>
</tbody>
</table>

Error Returns

[EBUSY] Other opens are active on the level for which exclusive access is desired (lun, target, bus).

[EINVAL] int is not in the range 0 to 5.

SIOC_XSENSE

The last available sense data is copied to the passed sense_data structure. It may be used by either a device driver or the pass-through driver.

Sense in the "lun" structure utilizes an array of 2 so that command-mode applications can get data associated with command-mode I/Os and not get misleading data from non-command-mode I/Os that happen to get interleaved with command-mode I/Os. x[CMD_MODE_ONLY] is updated for command-mode only I/Os while x[EVERYTHING] is updated for all I/Os including command-mode I/Os. This is managed automatically by SCSI Services depending upon whether the request is originating from the dev_t set to SCSI_CMD_MODE.
The sense_2, sense_2_aligned, sense_data, xsense, and xsense_aligned structures and other data are defined in the ../h/scsi.h header file.

### Error Returns

**[EINVAL]** The data size is not equal to 128 bytes.

**SIOC_IO**

ioctl(fd, SIOC_IO, &sctl_io)

Used for "pass-through" I/Os. The bp->b2_flags is used for B2_FIRST, for dissort(), and B2_LOWPRIO, for kmetrics, used for scsi_cmd(), ioctl(SIOC_IO) and ioctl(SIOC_CMD_MODE) type I/Os.

See the scsi_ctl(7) manpage.

It uses the lun open/close blocking semaphore to prevent a change in the status of concurrent non-pass-through opens to this LUN. This semaphore protects the lp->ddsw and lp->state value L_DD_PASS_THRU_VALID from being modified.

It must be noted that the sctl_io data structure used by SIOC_IO is passed into the driver by way of physio(). This data structure itself contains pointers to other data buffers. This violates the ioctl(2) manpage, which states, "Note: Any data structure referenced by arg must not contain any pointers." This "violation" is allowable because the driver and the SCSI Subsystem expect these pointers to exist in this structure. It must be noted that a future implementation for which physio() needs to map pointers — to other hosts’ memory for example — will not work for this case. Currently, there are no plans for such implementations.

Data transfer size maximum is SCSI_MAXPHYS, 1024 * 1024, or 1 megabyte.

### Error Returns

**[EACCES]** The user is not superuser or there is no write access permission.

**SIOC_PRIORITY_MODE**

ioctl(fd, SIOC_PRIORITY_MODE, &int)

See the scsi_ctl(7) manpage.

A device can only be put into priority-mode from the pass-through driver. Once in priority-mode, all pass-through driver SIOC_IO requests to the device are priority-mode I/Os; all other I/Os (not yet queued by SCSI Services in its scb queue) are blocked until the device is taken out of priority-mode. Also while in priority-mode, all device open attempts via the pass-through driver fail. Priority-mode poses a potential deadlock
problem. If the process which has a device in priority-mode blocks waiting for a non-priority-mode I/O to that same device, the result is deadlock. No other I/O to that device will occur — ever. Therefore, the process simply cannot do non-priority-mode I/O to the priority-mode device. Nor can the process allow the system to block it waiting for a page-fault or swap I/O to the device.

If not superuser, it returns \[EACCESS\].

The int parameter is an integer: The value 1 enables priority mode. The value 0 disables priority mode.

Error Returns

\[EBUSY\] The pass-through driver open count is not one.

\[EINVAL\] int is invalid, or the command was not invoked by the pass-through major number.

**SIOC_CMD_MODE**

\[ioctl(fd, SIOC_CMD_MODE, &int)\]

This may be used by either the device driver or the pass-through driver, sctl.

The int parameter can be either 0 (off) or 1 (on).

The bp->b2_flags is used for B2_FIRST, for disksort(), and B2_LOWPRIO, for kmetrics, used for scsi_cmd(), ioctl(SIOC_IO) and ioctl(SIOC_CMD_MODE) type I/Os.

Error Returns

\[EACCESS\] The caller is not superuser and the open was not with FWRITE.

\[EBUSY\]

- The pass-through driver is the caller and either it currently has more than one open or the device driver is already open.
- A device driver (raw) is the caller and it has more than one open currently.
- Any driver is the caller, command mode is already on, and this is not the driver that turned it on.

\[EINVAL\] int is invalid.

**SIOC_SET_CMD**

\[ioctl(fd, SIOC_SET_CMD, &scsi_cmd_parms)\]

This command may be used by either a device driver or the pass-through driver, sctl, so long as the lp->cmd_mode_major is the calling driver.
The structure `scsi_cmd_parms` is defined in `../h/scsi.h`.

**Error Returns**

- **[EACCES]** The command was not invoked by the "command code major".
- **[EINVAL]** The parameter, `scsi_cmd_parms->cmd_type`, is less than 1 or greater than `SCSI_MAX_CDB_LEN`.

**SIOC_RETURN_STATUS**

`ioctl(fd, SIOC_RETURN_STATUS, &int)`

May be used by either device driver or sctl pass-through driver, whether in command mode or not. The SCSI status will be that of the last I/O [EVERYTHING], or that of the last cmd_mode_major originated I/O [CMD_MODE_ONLY], unless "STINGRAY" (includes SCSI Cascade), for which `if2_x_status` is returned.

- `cdb_status[EVERYTHING]` set at every I/O completion
- `cdb_status[CMD_MODE_ONLY]` set at cmd_mode_major I/Os only
- `if2_last_status` set at every I/O completion
- `if2_scsi_status` set at every I/O when `cdb_status` least significant byte has bits on (i.e., is SCSI status; doesn't include SCTL_xxx).

This has special support for the STINGRAY/SCSI Cascade disk arrays, which rely upon SCSI Common Access Method (CAM) status for their utilities. In IF2, `SIOC_RETURN_STATUS` returned a combination of CAM status and SCSI status. The CAM status was from the most recent I/O and the SCSI status was from the most recent I/O with CAM status of `CS_REQ_COMP_WERROR`. The DMD/Stingray folks have been adamant about preserving this behavior; the mechanism is `(lp->state &L_IF2_STATUS)`.

For `SIOC_RETURN_STATUS` ioctl, status in the "lun" structure utilizes an array of 2 so that command-mode applications can get data associated with command-mode I/Os and not get misleading data from non-command-mode I/Os that happen to get interleaved with command-mode I/Os. `x[CMD_MODE_ONLY]` is updated for command-mode only I/Os while `x[EVERYTHING]` is updated for all I/Os including command-mode I/Os. This is managed automatically by SCSI Services, depending upon whether the request is originating from the `dev_t` set to `SCSI_CMD_MODE`.

The following is the code which supports this functionality in the SCSI Services:
/*
** CAM status values for backward compatibility.
** Pre-shifted for convenience.
** From wsio/scsi_ctl.h.
*/
#define CS_GOOD   (0x01 << 8)
#define CS_ABORTED_BY_HOST (0x02 << 8)
#define CS_REQ_COMP_WERROR (0x04 << 8)
#define CS_INVALID_REQUEST (0x06 << 8)
#define CS_SELECT_TIMEOUT (0x0a << 8)

if (!pass_thru_major(dev) && (lp->state & L_IF2_STATUS))
{
    i = lp->if2_last_status;
    j = i == SCTL_SELECT_TIMEOUT ? CS_SELECT_TIMEOUT
        : i == SCTL_INVALID_REQUEST
        ? CS_INVALID_REQUEST
        : i == S_GOOD
        ? CS_GOOD
        : CS_REQ_COMP_WERROR;
    k = j | lp->if2_scsi_status;
} else
{
    /* Non "STINGRAY" case */
    i = major(dev) == lp->cmd_mode_major
        ? CMD_MODE_ONLY : EVERYTHING;
    /* 1:0 */
    k = lp->cdb_status[i];
    lp->cdb_status[i] = -1;
}
*(u_int *) data = k;
return 0;

Error Returns
None.

SIIOC_GET_LUN_PARMS
ioctl(fd, SIIOC_GET_LUN_PARMS, &sioc_lun_parms)
The structure sioc_lun_parms is defined in ../h/scsi.h. See scsi_ctl(7).

Error Returns
None.

SIIOC_GET_TGT_PARMS
ioctl(fd, SIIOC_GET_TGT_PARMS, &sioc_tgt_parms)
The structure `sioc_tgt_parms` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

None.

**SIOC_GET_BUS_PARMS**

`ioctl(fd, SIOC_GET_BUS_PARMS, &sioc_bus_parms)`

The structure `sioc_bus_parms` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

None.

**SIOC_GET_LUN_LIMITS**

`ioctl(fd, SIOC_GET_LUN_LIMITS, &sioc_lun_limits)`

If limits have not been set, the act of getting them, sets them.

The structure `sioc_lun_limits` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

None.

**SIOC_GET_TGT_LIMITS**

`ioctl(fd, SIOC_GET_TGT_LIMITS, &sioc_tgt_limits)`

If limits have not been set, the act of getting them, sets them.

The structure `sioc_tgt_limits` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

None.

**SIOC_GET_BUS_LIMITS**

`ioctl(fd, SIOC_GET_BUS_LIMITS, &sioc_bus_limits)`

If limits have not been set, the act of getting them, sets them.

The structure `sioc_bus_limits` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

None.

**SIOC_SET_LUN_LIMITS**

`ioctl(fd, SIOC_SET_LUN_LIMITS, &sioc_lun_limits)`

The structure `sioc_lun_limits` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

[EACCES] If not superuser or write permission.

[EINVAL] If reserved fields are not zero.
SCSI Functions

**SCSI_SET_TGT_LIMITS**

```c
ioctl(fd, SIOC_SET_TGT_LIMITS, &sioc_tgt_limits)
```

The structure `sioc_tgt_limits` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

- `[EACCES]` If not superuser or write permission.
- `[EINVAL]` If reserved fields are not zero.

**SCSI_SET_BUS_LIMITS**

```c
ioctl(fd, SIOC_SET_BUS_LIMITS, &sioc_bus_limits)
```

The structure `sioc_bus_limits` is defined in `../h/scsi.h`. See `scsi_ctl(7)`.

**Error Returns**

- `[EACCES]` If not superuser or write permission.
- `[EINVAL]` If reserved fields are not zero.

**SCSI_RESET_DEV**

```c
ioctl(fd, SIOC_RESET_DEV)
```

This command causes a SCSI Bus Device Reset to be sent to the target device by calling the Interface Driver’s `if_bdr()` routine.

**Error Returns**

- `[EACCES]` The user is not superuser.
- `[EINVAL]` `if_bdr()` is NULL.

**SCSI_RESET_BUS**

```c
ioctl(fd, SIOC_RESET_BUS)
```

This command causes the SCSI RST line to be "pulled," by calling the Interface Driver’s `if_reset_bus()` routine.

**Error Returns**

- `[EACCES]` The user is not superuser.
- `[EINVAL]` `if_reset_bus()` is NULL.
NAME

scsi_lun_close(SCSI3)  SCSI Function – Close a device

SYNOPSIS

scsi_lun_close (dev_t dev);

PARAMETERS

dev  The device number

DESCRIPTION

The scsi_lun_close() function is called to close a device. It must be called in the process context and may block. It is not called from within any critical section. It uses the lun open/close semaphore.

scsi_lun_close() performs the following algorithm:

• Acquire the logical unit open/close semaphore.
  — If this is the last nonpass-through close, wait for all nonpass-through I/O's to complete.
  — If this is any nonpass-through close, call device driver close routine, dd_close().
  — If this is the last nonpass-through close, clear (lp->state &L_NPT_DD_BITS).
  — Attempt to honor the new state with respect to tagged queuing.
  — Call the interface driver close routine, ifsw->if_close.
  — Update logical unit open counts.
  — Exit command-mode if closing process neglected to do so.
  — Exit priority-mode if closing process neglected to do so.
  — If this is the last nonpass-through close:
    — Free *lp->dd_lun and clear lp->dd_lun if necessary.
    — lp->ddsw = NULL.
    — If this is the last close:
— Free any logical unit SCBs that may be hanging around.
— Free *lp->if_lun and *lp and clear tp->lun[lun_id].

- Release the logical unit semaphore.
- Update the ancestor portion of the open device tree via `scsi_tgt_close()`.

This routine assumes that `dev` is open. If it is not, the system will panic in `m_scsi_lun()` when it attempts to dereference a NULL pointer.

**RETURN VALUES**

`scsi_lun_close()` returns an indeterminate value. Treat it as if it was a `void` function.
NAME

```c
scsi_lun_open(SCSI)  SCSI Function – Open the elements of the hardware
path of a SCSI Lun
```

SYNOPSIS

```c
scsi_lun_open (dev_t dev, struct scsi_ddsw * dds, int oflags);
```

PARAMETERS

- `dev` The device number
- `ddsw` Pointer to the non pass_thru driver descriptor
- `oflags` File access flags

DESCRIPTION

Usually called from the device driver's `driver_dev_init()` routine, The
`scsi_lun_open()` SCSI function performs necessary open operations
down the hardware path upon which this SCSI LUN resides, including
the invocation of the calling driver's `ddsw->dd_open()` routine. It opens
the appropriate target if necessary. Also used by the pass-through driver.

Used by device drivers, this function must be called in the process
context and may block. The function is not called from within any critical
section. It verifies that no spinlocks are held with call to `SD_ASSERT`. 
Does use the lun open/close semaphore.

Uses `kmalloc()` to allocate memory for the `scsi_lun` structure.
Checks if `major(dev) == sctl_ddsw.raw_major`.

RETURN VALUES

- `scsi_lun_open()` returns the following values:
  - 0  Successful completion.
  - [EBUSY] The LUN is already opened EXCLUSIVE by another or
    the open is incomplete and this isn't a pass-through or
    command mode open attempt.
  - [EINVAL] The open request major number doesn't make sense.
The LUN requested is greater than `SCSI_MAX_LUN_ID`.

Other errors may be returned from `ddsw->dd_open()`, `if_open()`, `scsi_bus_open()`, or `scsi_tgt_open()`, if they are called from here.
NAME

scsi_read(SCSI3) SCSI Function - Read from device

SYNOPSIS

scsi_read(dev_t dev, struct uio *uio);

PARAMETERS

dev The device number
uio struct containing transfer information

DESCRIPTION

The scsi_read() SCSI function is used for normal (synchronous) reads, and for command mode I/Os for which the ioctl, SCSI_CMD_MODE, has been set previously. For normal I/Os, if the driver has defined a dd_read() routine in the scsi_ddsw structure, it is called; otherwise, physio() is called directly.

Used by device drivers, this function must be called in the process context and may block. The function is not called from within any critical section.

RETURN VALUES

scsi_read() returns the following values:

0 Successful completion.
<>0 Error. The value is expected to be an errno value.

SEE ALSO

physio(KER2), scsi_ddsw(SCSI4)
NAME

scsi_sense_action(SCSI) SCSI Function – Decode SCSI sense information

SYNOPSIS

scsi_sense_action (struct buf *bp,
               struct sense_action *sense_list, int n)

PARAMETERS

bp Pointer to the I/O buf structure
n Number of actions in the list.
sense_list List of actions to take.

DESCRIPTION

The scsi_sense_action() SCSI function decodes SCSI sense information. It traverses the functions in a driver’s sense action list trying to find a match, and calls the associated action function. It provides the very valuable service of interpreting sense data with regard to SCSI, CCS, or SCSI-2 compliance, so the device driver doesn’t need to worry about such things.

scsi_sense_action() should be called only in the interrupt context by device drivers and by SCSI services on behalf of a device driver.

This function only operates on the request. Sense information does not appear to have any real protection concerns.

The inquiry data for the device must be initialized with scsi_init_inquiry_data() before it can be interpreted. If an I/O completes before the inquiry data is initialized, scsi_sense_action() will not match anything other than wild card entries. It will panic if there is no matching entry.

RETURN VALUES

scsi_sense_action() returns the following values:

0 Successful completion.
Error. The value is provided by the sense action called.

SEE ALSO

scsi_init_inquiry_data(SCSI3)
NAME

scsi_strategy(SCSI3) SCSI Function - scsi_strategy -

SYNOPSIS

void scsi_strategy (struct buf * bp)

PARAMETERS

bp The pointer to the I/O buf structure.

DESCRIPTION

The scsi_strategy() SCSI function primarily enqueues the bp to await the necessary resources to allow the request to be sent to the interface driver, and thus, the hardware.

Another purpose is to record the fact that an I/O has been enqueued so the device is not closed while unfinished I/Os exist.

This routine is the first place in the I/O path that all I/Os have in common.

scsi_strategy() is usually called in the process context; it may be invoked on the interrupt context (possibly in the case of a bp->b_call used by the biodone() of a previous I/O completion). Regardless, scsi_strategy() cannot block. Verifies no spinlocks are held by calling SD_ASSERT.

scsi_strategy() must be invoked with a valid bp. If a “special” request (i.e., either B_SIOC_IO or B_SCSI_CMD), it calls scsi_enqueue() to place bp in the lp->priority_scb_q (if lp->pri_mode_major) or lp->special_scb_q. Otherwise, it calls the dd_strategy() routine so that the device driver can manage request order, etc. In this last (normal I/O) case, bp->b2_flags has B2_LOWPRIO cleared for kmetrics' support. Finally, it calls scsi_start().

NOTE

scsi_strategy() calls dd_strategy(), if present, holding the lun_lock.
RETURN VALUES

`scsi_strategy()` is a `void` function.

SEE ALSO

`biodone(KER2), scsi_enqueue(SCSI3)`
NAME

scsi_write(SCSI3) SCSI Function – Write to device

SYNOPSIS

scsi_write (dev_t dev, struct uio * uio);

PARAMETERS

dev The device number
uio struct containing transfer information

DESCRIPTION

The scsi_write() SCSI function is used for normal (synchronous) writes and for command mode I/Os for which the ioctl, SCSI_CMD_MODE, has been set previously. For normal I/Os, if the driver has defined a dd_write() routine in the scsi_ddsw structure, it is called; otherwise, physio() is called directly.

Used by device drivers, this function must be called in the process context and may block. The function is not called from within any critical section.

RETURN VALUES

scsi_write() returns the following values:

0 Successful completion.
<>0 Error. The value is expected to be an errno value.

SEE ALSO

physio(KER2), scsi_ddsw(SCSI4)
Index

A
alloc_spinlock() kernel function, 20

dd_start() SCSI functions, 404
dd_strategy() SCSI function, 405
dd_write() SCSI function, 406
dealloc_spinlock() kernel function, 55
determine interrupt assigned to card, 257
device file ISC structure pointer, 259
device probe structure, 279
disclaimers
  support/compatibility, 14
dma_cleanup() WSIO function, 188
dma_parms CDIO structure, 189
dma_setup() WSIO function, 191
dma_sync() CDIO macro, 168, 171
driver_addr_probe() WSIO function, 192
driver_attach() WSIO function, 194
driver_class_probe() WSIO function, 196
driver_close() WSIO function, 198
driver_dev_init() WSIO function, 200
driver_if_init() WSIO function, 201
driver_install() WSIO function, 202
driver_ioctl() WSIO function, 204
driver_isr() WSIO function, 206
driver_minor_build() WSIO function, 207
driver_minphys() WSIO function, 209
driver_open() WSIO function, 210
driver_psize() WSIO function, 212
driver_read() WSIO function, 213
driver_select() WSIO function, 215
driver_strategy() WSIO function, 217
driver_write() WSIO function, 218
driver-specific probing function, 192, 196
drv_info CDIO structure, 174
drv_ops CDIO structure, 177

B
b_cpsema() kernel function, 22
b_initsema() kernel function, 25
b_owns_sema() kernel function, 27
b_psem() kernel function, 29
b_vsema() kernel function, 31
bcmp() kernel function, 33
bcopy() kernel function, 34
beta semaphore kernel functions, 29, 31
Beta Semaphores, 22, 25, 27, 29, 31
biodone() kernel function, 36
biowait() kernel function, 38
brelse() kernel function, 40
buf kernel structure, 41
build minor number, 207
busywait() kernel function, 46
bzero() kernel function, 47

card, interrupt assigned, 257
CDIO functions
  install_driver(), 183
CDIO macros
  dma_sync(), 168, 171
  init_map_context(), 181
CDIO structures
  dma_parms, 189
drv_info, 174
drv_ops, 177
claim ISC structure element, 227
Coherency Issues, memory, 388
Config, Cycle Retry Problem, PCI, 392
CONNECT_INIT_ROUTINE() PCI macro, 363
copyin() kernel function, 48
copyout() kernel function, 50
cspinlock() kernel function, 52

dd_close() SCSI function, 396
dd_done() SCSI function, 397
dd_ioctl() SCSI function, 398
dd_ioctlOkay() SCSI function, 399
dd_open() SCSI function, 400
dd_pass_thru_done() SCSI function, 401
dd_pass_thruOkay() SCSI function, 402
dd_read() SCSI function, 403

E
erata PCI, 388, 391, 392
ETHER_MAP_IP_MULTICAST Network macro, 302

F
FREE() kernel macro, 56, 84
free_isc() WSIO function, 220
function
  dd_open(), 400
functions
  alloc_spinlock(), 20
  b_cpsema(), 22
  b_initsema(), 25
  b_owns_sema(), 27
  b_psem(), 29
  b_vsema(), 31
  bcmp(), 33

443
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcopy()</td>
<td>34</td>
</tr>
<tr>
<td>beta semaphore</td>
<td>29, 31</td>
</tr>
<tr>
<td>biodone()</td>
<td>36</td>
</tr>
<tr>
<td>brelse()</td>
<td>38</td>
</tr>
<tr>
<td>busywait()</td>
<td>40</td>
</tr>
<tr>
<td>bzero()</td>
<td>46</td>
</tr>
<tr>
<td>copyin()</td>
<td>47</td>
</tr>
<tr>
<td>copyout()</td>
<td>48</td>
</tr>
<tr>
<td>csinlock()</td>
<td>52</td>
</tr>
<tr>
<td>dd_close()</td>
<td>396</td>
</tr>
<tr>
<td>dd_done()</td>
<td>397</td>
</tr>
<tr>
<td>dd_ioctl()</td>
<td>398</td>
</tr>
<tr>
<td>dd_ioctl_okay()</td>
<td>399</td>
</tr>
<tr>
<td>dd_pass_thru_done()</td>
<td>401</td>
</tr>
<tr>
<td>dd_pass_thru_okay()</td>
<td>402</td>
</tr>
<tr>
<td>dd_read()</td>
<td>403</td>
</tr>
<tr>
<td>dd_start()</td>
<td>404</td>
</tr>
<tr>
<td>dd_strategy()</td>
<td>405</td>
</tr>
<tr>
<td>dd_write()</td>
<td>406</td>
</tr>
<tr>
<td>dealloc_spinlock()</td>
<td>55</td>
</tr>
<tr>
<td>dma_cleanups()</td>
<td>188</td>
</tr>
<tr>
<td>dma_setup()</td>
<td>191</td>
</tr>
<tr>
<td>driver_addr_probe()</td>
<td>192</td>
</tr>
<tr>
<td>driver Attach()</td>
<td>194</td>
</tr>
<tr>
<td>driver_class_probe()</td>
<td>196</td>
</tr>
<tr>
<td>driver_close()</td>
<td>198</td>
</tr>
<tr>
<td>driver_dev_init()</td>
<td>200</td>
</tr>
<tr>
<td>driver_if_init()</td>
<td>201</td>
</tr>
<tr>
<td>driver_install()</td>
<td>202</td>
</tr>
<tr>
<td>driver_ioctl()</td>
<td>204</td>
</tr>
<tr>
<td>driver_isr()</td>
<td>206</td>
</tr>
<tr>
<td>driver_minor_build()</td>
<td>192</td>
</tr>
<tr>
<td>driver_minphys()</td>
<td>209</td>
</tr>
<tr>
<td>driver_open()</td>
<td>210</td>
</tr>
<tr>
<td>driver_psize()</td>
<td>212</td>
</tr>
<tr>
<td>driver_read()</td>
<td>213</td>
</tr>
<tr>
<td>driver_select()</td>
<td>215</td>
</tr>
<tr>
<td>driver_strategy()</td>
<td>217</td>
</tr>
<tr>
<td>driver_write()</td>
<td>218</td>
</tr>
<tr>
<td>free_isc()</td>
<td>220</td>
</tr>
<tr>
<td>get_new_isc()</td>
<td>221</td>
</tr>
<tr>
<td>get_sleep_lock()</td>
<td>58</td>
</tr>
<tr>
<td>getc()</td>
<td>60</td>
</tr>
<tr>
<td>getcb()</td>
<td>62</td>
</tr>
<tr>
<td>getcf()</td>
<td>64</td>
</tr>
<tr>
<td>geteblk()</td>
<td>65</td>
</tr>
<tr>
<td>geterror()</td>
<td>67</td>
</tr>
<tr>
<td>gsignal()</td>
<td>69</td>
</tr>
<tr>
<td>hw_if_attach()</td>
<td>308</td>
</tr>
<tr>
<td>HWIFT_LOCK()</td>
<td>309</td>
</tr>
<tr>
<td>HWIFT_UNLOCK()</td>
<td>309</td>
</tr>
<tr>
<td>install_driver()</td>
<td>183</td>
</tr>
<tr>
<td>iodone()</td>
<td>223</td>
</tr>
<tr>
<td>iowait()</td>
<td>225</td>
</tr>
<tr>
<td>isc_dain()</td>
<td>227</td>
</tr>
<tr>
<td>isrlink()</td>
<td>229</td>
</tr>
<tr>
<td>isrunlink()</td>
<td>231</td>
</tr>
<tr>
<td>kernel_iomap()</td>
<td>77</td>
</tr>
<tr>
<td>kernel_iomap_public()</td>
<td>80</td>
</tr>
<tr>
<td>kernel_iounmap()</td>
<td>83</td>
</tr>
<tr>
<td>kget_log_instance()</td>
<td>311</td>
</tr>
<tr>
<td>KLOGCK()</td>
<td>312</td>
</tr>
<tr>
<td>klogg_write()</td>
<td>314</td>
</tr>
<tr>
<td>kmalloc()</td>
<td>85</td>
</tr>
<tr>
<td>Ktimeout()</td>
<td>87</td>
</tr>
<tr>
<td>KTRC_CHECK()</td>
<td>316</td>
</tr>
<tr>
<td>ktrc_write()</td>
<td>318</td>
</tr>
<tr>
<td>m_scsi_lum()</td>
<td>407</td>
</tr>
<tr>
<td>m_wsio_funcnum()</td>
<td>234</td>
</tr>
<tr>
<td>m_wsio_selcode()</td>
<td>235</td>
</tr>
<tr>
<td>m_wsio_vsc()</td>
<td>236</td>
</tr>
<tr>
<td>map_mem_to_host()</td>
<td>96</td>
</tr>
<tr>
<td>minphys()</td>
<td>99</td>
</tr>
<tr>
<td>mod_wsio_attach_list_add()</td>
<td>237</td>
</tr>
<tr>
<td>mod_wsio_attach_list_remove()</td>
<td>239</td>
</tr>
<tr>
<td>msg_printf()</td>
<td>101</td>
</tr>
<tr>
<td>p_pgrp()</td>
<td>105</td>
</tr>
<tr>
<td>panic()</td>
<td>107</td>
</tr>
<tr>
<td>pci_desc_bus_transactions_isc()</td>
<td>365</td>
</tr>
<tr>
<td>pci_get_fru_info_isc()</td>
<td>367</td>
</tr>
<tr>
<td>pci_get_port_hndl_isc()</td>
<td>369</td>
</tr>
<tr>
<td>pci_read_cfg_uintN_isc()</td>
<td>371</td>
</tr>
<tr>
<td>pci_read_port_uintN_isc()</td>
<td>373</td>
</tr>
<tr>
<td>pci_unget_port_hndl_isc()</td>
<td>376</td>
</tr>
<tr>
<td>pci_write_cfg_uintN_isc()</td>
<td>378</td>
</tr>
<tr>
<td>pci_write_port_uintN_isc()</td>
<td>380</td>
</tr>
<tr>
<td>physio()</td>
<td>108</td>
</tr>
<tr>
<td>printf()</td>
<td>112</td>
</tr>
<tr>
<td>privlbcopy()</td>
<td>89, 114</td>
</tr>
<tr>
<td>psignal()</td>
<td>116</td>
</tr>
<tr>
<td>putc()</td>
<td>118</td>
</tr>
<tr>
<td>putcb()</td>
<td>120</td>
</tr>
<tr>
<td>putcf()</td>
<td>121</td>
</tr>
<tr>
<td>READ_REG_UINTN_ISC()</td>
<td>382</td>
</tr>
<tr>
<td>scsi_action()</td>
<td>408</td>
</tr>
<tr>
<td>scsi_cmd()</td>
<td>411</td>
</tr>
<tr>
<td>scsi_cmds()</td>
<td>413</td>
</tr>
</tbody>
</table>
Index

scsi_ddsw(), 415
scsi_dequeue(), 419
scsi_dequeue_bp(), 421
scsi_enqueue(), 422
scsi_init_inquiry_data(), 423
scsi_ioctl(), 424
scsi_lun_close(), 432
scsi_lun_open(), 434
scsi_read(), 436
scsi_sense_action(), 437
scsi_strategy(), 439
scsi_write(), 441
selwakeup(), 123
set_up_8022(), 320
set_up_ether(), 322
set_up_ip(), 324
set_up_link(), 326
sleep(), 125
spinlock(), 55, 58, 103, 128
spinunlock(), 130
sprintf(), 132
strcat(), 134
strcmp(), 135
strcpy(), 137
strlen(), 139
strncmp(), 141
strncpy(), 143
subsys_N_format(), 328
subsys_N_get_options(), 334
suser(), 145
sw_trigger(), 146
timeout(), 149
tl_banner_char(), 338
tl_check_cat_version(), 339
tl_format_fprintf(), 341
tl_format_write(), 343
tl_get_line(), 346
tl_get_parms(), 348
 tl_header_format1(), 352
 tl_log_dasses(), 355
tl_rae_format(), 357
tl_trace_kind(), 359
uiomove(), 153
unmap_mem_from_host(), 156
untimeout(), 157
user_iomap(), 158
user_iounmap(), 161
VASSERT(), 162
wakeup(), 163
WRITE_REG_UINTn_ISC(), 385
wsio_activate_probe(), 241
wsio_allocat_shared_memory(), 244
wsio_fastmap(), 250
wsio_flush_shared_memory(), 253
wsio_free_shared_memory(), 255
wsio_get_interrups(), 257
wsio_get_isc(), 259
wsio_get_pva(), 261
wsio_get_registers(), 263
wsio_install_driver(), 265
wsio_isc_to_instance(), 268
wsio_map(), 270
wsio_register_addr_probe(), 281
wsio_register_devr_probe(), 284
wsio_register_probe_func(), 287
wsio_remap(), 289
wsio_set_attributes(), 291
wsio_uninstall_driver(), 293
wsio_unmap(), 294
wsio_unregister_dev_probe(), 296
G
get register addresses of interface card, 263
get_new_isc() WSIO function, 221
get_sleep_lock() kernel function, 58
getc() kernel function, 60
getcb() kernel function, 62
getcf() kernel function, 64
getblk() kernel function, 65
gterror() kernel function, 67
gsignal() kernel function, 69
H
hw_ift() Network structure, 303
hw_ift_attach() Network function, 308
HWIFT_LOCK() Network function, 309
HWIFT_UNLOCK() Network function, 309
I
init_map_context() CDIO macro, 181
install_driver() CDIO function, 183
instance number of iotree node, 268
intended audience, 13
interface card register addresses, 263
interface select code table, 72
interrupt assigned to card, 257
interrupt service routine, 206
iodone() WSIO function, 223
iotree node instance number, 268
Index

iovec kernel structure, 71
iowait() WSIO function, 225
ISC structure element, claim, 227
ISC structure pointer for device file, 259
isc CLAIM WSIO function, 227
isc_table_type kernel structure, 72
isrlink() WSIO function, 229
isrunlink() WSIO function, 231
K
kernel functions
  alloc_spinlock(), 20
  b_cpsema(), 22
  b_initsema(), 25
  b_lists_sema(), 27
  b_psema(), 29
  b_vsema(), 31
  bcmpl(), 33
  bcopy(), 34
  beta semaphore, 29, 31
  biodone(), 36
  biowait(), 38
  brelse(), 40
  busywait(), 46
  bzerol(), 47
  copyin(), 48
  copyout(), 50
  cspinlock(), 52
  dealloc_spinlock(), 55
  get_sleep_lock(), 58
  getc(), 60
  getcbl(), 62
  getcf(), 64
  geteblk(), 65
  geterror(), 67
  gsingal(), 69
  kernel iomap(), 77
  kernel_iomap_public(), 80
  kernel_iounmap(), 83
  kmalloc(), 85
  Ktimeout(), 87
  map_mem_to_host(), 96
  minphys(), 99
  msg_printf(), 101
  p_pgrp(), 105
  panic(), 107
  physid(), 108
  printf(), 112
  privlbcopy(), 89, 114
  psignal(), 116
  putc(), 118
  putcb(), 120
  putcf(), 121
  selwakeup(), 123
  sleep(), 125
  spinlock(), 55, 58, 103, 128
  spinunlock(), 130
  sprintf(), 132
  strcat(), 134
  strcmp(), 135
  strcpy(), 137
  strlen(), 139
  strncmp(), 141
  strncpy(), 143
  suser(), 145
  sw_trigger(), 146
  timeout(), 149
  uiomove(), 153
  unmmap_mem_from_host(), 156
  untimeout(), 157
  user_iomap(), 158
  user_iounmap(), 161
  VASSERT(), 162
  wakeup(), 163
kernel macros
  FREE(), 56, 84
  major(), 91
  makedev(), 92
  MALLOCl(), 93
  minor(), 98
kernel structures
  buf, 41
  iovec, 71
  isc_table_type, 72
  ui0, 151
kernel iomap() kernel function, 77
kernel_iomap_public() kernel function, 80
kernel_iounmap kernel function, 83
kget_log_instance() Network function, 311
KLOG CK() Network function, 312
klog_write() Network function, 314
kmalloc() kernel function, 85
Ktimeout() kernel function, 87
KTRC_CHECK() Network function, 316
ktrc_write() Network function, 318
M
  m_instance() WSIO macro, 233
  m_scsi_lum() SCSI function, 407
m_wsio_funcnum() WSIO function, 234
m_wsio_selcode() WSIO function, 235
m_wsio_vsc() WSIO function, 236

macros
CONNECT_INIT_ROUTINE(), 363
dma_sync(), 168, 171
ETHER_MAP_IP_MULTICAST, 302
FREE(), 56, 84
init_map_context(), 181
m_instance(), 233
major(), 91
makedev(), 92
MALLOC(), 93
minor(), 98
PCI_ATTACH_DEV_INIT_ERROR(), 364
major() kernel Macro, 91
makedev() kernel Macro, 92
MALLOC() kernel macro, 93
manpage, topic format, 15
map_mem_to_host() kernel function, 96
mark ISC Structure element as claimed by
driver, 227
Memory Coherency Issues, 388
minor number, build, 207
minor() kernel macro, 98
minphys() kernel function, 99
mod_wsio_attach_add() WSIO function, 237
mod_wsio_attach_list_remove() WSIO
function, 239
msg_printf() kernel function, 101

N
Network functions
hw_if_attach(), 308
HWIFIT_LOCK(), 309
HWIFIT_UNLOCK(), 309
kget_log_instance(), 311
KLOG_CK(), 312
klogwrit(), 314
KTRC_CHECK(), 316
ktrc_write(), 318
set_up_8022(), 320
set_up_ether(), 322
set_up_ipl(), 324
set_up_link(), 326
subsys_N_format(), 328
subsys_N_get_options(), 334
tl_banner_char(), 338
tl_check_Cat_version(), 339
tl_format_fprintf(), 341
tl_format_write(), 343
tl_get_line(), 346
tl_get_parms(), 348
tl_header_format1(), 352
tl_log_classes(), 355
tl_rae_format(), 357
tl_trace_kind(), 359

Network macros
ETHER_MAP_IP_MULTICAST, 302
Network structures
hw_if_t(), 303

P
p_ggrp() kernel function, 105
panic() kernel function, 107
PCI Config. Cycle Retry Problem, 392
PCI errata, 388, 391, 392
PCI functions
pci_desc_bus_transactions_iscl(), 365
pci_get_fru_info_iscl(), 367
pci_get_port_hndl_iscl(), 369
pci_read_cfg_uintN_iscl(), 371
pci_read_port_uintN_iscl(), 373
pci_unget_port_hndl_iscl(), 376
pci_write_cfg_uintN_iscl(), 378
pci_write_port_uintN_iscl(), 380
READ_REG_UINTn_ISC(), 382
WRITE_REG_UINTn_ISC(), 385
PCI macros
CONNECT_INIT_ROUTINE(), 363
PCI_ATTACH_DEV_INIT_ERROR(), 364
PCI Transaction Ordering Errata, 391, 392
PCI_ATTACH_DEV_INIT_ERROR() PCI
macro, 364
pci_desc_bus_transactions_iscl() PCI
function, 365
pci_get_fru_info_iscl() PCI function, 367
pci_get_port_hndl_iscl() PCI function, 369
pci_read_cfg_uintN_iscl() PCI functions, 371
pci_read_port_uintN_iscl() PCI functions, 373
pci_unget_port_hndl_iscl() PCI function, 376
pci_write_cfg_uintN_iscl() PCI functions, 378
pci_write_port_uintN_iscl() PCI functions,
380
physio() kernel function, 108
printf() kernel function, 112
privlbcopy() kernel function, 89, 114
probe structure, device, 279
probing function, 192, 196
psignal() kernel function, 116
putc() kernel function, 118
putcb() kernel function, 120
putcf kernel function, 121

R
READ_REG_UINTn_ISC() PCI functions, 382
register addresses of interface card, 263
retrieve instance number of iotree node, 268
retrieve ISC structure pointer for device file, 259
Retry Problem, Config. Cycle PCI, 392
routines
  character device read/write, 213, 218

S
SCSI functions
dd_close(), 396
dd_done(), 397
dd_ioctl(), 398
dd_ioctl_okay(), 399
dd_open(), 400
dd_pass_thru_done(), 401
dd_pass_thru_okay(), 402
dd_read(), 403
dd_start(), 404
dd_strategy(), 405
dd_write(), 406
m_scsi_lum(), 407
scsi_action(), 408
scsi_cmd(), 411
scsi_cmds(), 413
scsi_ddsw(), 415
scsi_dequeue(), 419
scsi_dequeue_bp(), 421
scsi_enqueue(), 422
scsi_init_inquiry_data(), 423
scsi_ioctl(), 424
scsi_lun_close(), 432
scsi_lun_open(), 434
scsi_read(), 436
scsi_read_complete(), 437
scsi_sense_action(), 438
scsi_strategy(), 439
scsi_write(), 441
scsi_init_inquiry_data() SCSI function, 423
scsi_ioctl() SCSI function, 424
scsi_lun_close() SCSI function, 432
scsi_lun_open() SCSI function, 434
scsi_read() SCSI function, 436
scsi_sense_action() SCSI function, 437
scsi_strategy() SCSI function, 439
scsi_write() SCSI function, 441

select code
  interface, 72
selwakeup() kernel function, 123
set_up_8022() Network function, 320
set_up_ether() Network function, 322
set_up_ip() Network function, 324
set_up_link() Network function, 326
sleep() kernel function, 125
software trigger, 146
spinlock() kernel function, 55, 58, 103, 128
Spinlocks, 20, 52, 55, 103, 128, 130
spinunlock() kernel function, 130
sprintf() kernel function, 132
strcat() kernel function, 134
strcmp() kernel function, 135
strcpy() kernel function, 137
strlen() kernel function, 139
strncmp() kernel function, 141
strncpy() kernel function, 143
structures, device probe, 279
structures
  buf, 41
dma_parms, 189
drv_info, 174
drv_ops, 177
hw_lft(), 303
iovec, 71
isc_table_type, 72
isc unix(), 151
wsio_drv_data, 247
wsio_drv_info, 249
wsio_probe_dev_info(), 279
subsys_N_format() Network function, 328
subsys_N_get_options() Network function, 334
support disclaimers, 14
suser() kernel function, 145
sw_trigger() kernel function, 146

T
timeout() kernel function, 149
tl_banner_char() Network function, 338
tl_check_cat_version() Network function, 339
tl_format_fprintf() Network function, 341
tl_format_write() Network function, 343
tl_get_line() Network function, 346
tl_get_parms() Network function, 348
tl_header_format1() Network function, 352
tl_log_classes() Network function, 355
tl_rae_format() Network function, 357
tl_trace_kind() Network function, 359

trigger, software, 146

u
uiomove() kernel function, 153
unmap_mem_from_host() kernel function, 156
untimeout() kernel function, 157
user_iomap() kernel function, 158
user_iounmap() kernel function, 161

V
VASSERT() kernel function, 162

W
wakeup() kernel function, 163
WRITE_REG_UINTn_ISC() PCI functions, 385
WSIO device probe structure, 279
WSIO functions
dma_cleanup(), 188
dma_setup(), 191
driver_addr_probe(), 192
driver_attach(), 194
driver_class_probe(), 196
driver_close(), 198
driver_dev_init(), 200
driver_if_init(), 201
driver_install(), 202
driver_ioctl(), 204
driver_isr(), 206
driver_minor_build(WSIO_DRV), 207
driver_minphys(), 209
driver_open(), 210
driver_psize(), 212
driver_read(), 213
driver_select(), 215
driver_strategy(), 217
driver_write(), 218
free_isc(), 220
get_new_isc(), 221
iodone(), 223
iowait(), 225
isc Claim(), 227
isrlink(), 229
isrunlink(), 231
m WSIO_funcnum(), 234
m WSIO_selcode(), 235
m WSIO_vsc(), 236
mod WSIO_attach_list_add(), 237
mod WSIO_attach_list_remove(), 239
wsio_activate_probe(), 241
wsio_allocat_shared_memory(), 244
wsio_free_shared_memory(), 255
wsio_flush_shared_memory(), 253
wsio_get_interrups(), 257
wsio_get_isc(), 259
wsio_get_pva(), 261
wsio_get_registers(), 263
wsio_install_driver(), 265
wsio_isc_to_instance(), 268
wsio_map(), 270
wsio_register_addr_probe(), 281
wsio_register_devr_probe(), 284
wsio_register_probe_func(), 287
wsio_remap(), 289
wsio_set_attributes(), 291
wsio_uninstall_driver(), 293
wsio_unmap(), 294
wsio_unregister_dev_probe(), 296
WSIO macros
m_instance(), 233
WSIO structures
wsio_drv_data, 247
wsio_drv_info, 249
wsio_iowait_dev_info(), 279
wsio_allocat_shared_memory() WSIO function, 244
wsio_drv_data WSIO function, 241
wsio_drv_data WSIO structure, 247
wsio_drv_info WSIO structure, 249
wsio_fastmap() WSIO function, 250
wsio_flush_shared_memory() WSIO function, 253
wsio_free_shared_memory() WSIO function, 255
wsio_get_interrups() WSIO function, 257
wsio_get_isc() WSIO function, 259
wsio_get_pva() WSIO function, 261
wsio_get_registers() WSIO function, 263
wsio_install_driver() WSIO function, 265
wsio_isc_to_instance() WSIO function, 268
Index

wsio_map() WSIO function, 270
wsio_probe_dev_info() WSIO structure, 279
wsio_register_addr_probe() WSIO function, 281
wsio_register_devr_probe() WSIO function, 284
wsio_register_probe_func() WSIO function, 287
wsio_remap() WSIO function, 289
wsio_set_attributes() WSIO function, 291
wsio_uninstall_driver() WSIO function, 293
wsio_unmap() WSIO function, 294
wsio_unregister_dev_probe() WSIO function, 296