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 incorporate all updates since the previous edition. The manual part number and printing date identify the current edition.

**Edition / Part Number / Date**

First Edition / B2355-90670 / September 1999
Second Edition / B2355-90726 / August 2000
Third Edition / B2355-90729 / November 2000
- Added Deprecated WSIO Interfaces Appendix, made minor technical and formatting changes throughout
Fourth Edition / B2355-90761 / July 2002
- Added Interrupt Migration manpage information and SCSI IOCTL updates. Other minor technical and formatting changes were also made throughout.
## Contents

1. Introduction
   - The Intended Audience ........................................... 15
   - Support/Compatibility Disclaimers ............................. 16
   - Reference Page Format ........................................... 17
   - Reference Page Sections ....................................... 18

2. Kernel Reference Pages
   - Functions and Structures ....................................... 21
     - alloc_spinlock(KER2) ......................................... 22
     - b_cpsema(KER2) ................................................ 24
     - b_initsema(KER2) ............................................... 27
     - b_owns_sema(KER2) .......................................... 29
     - b_psema(KER2) .................................................. 31
     - b_vsema(KER2) .................................................. 33
     - bcmp(KER2) ...................................................... 35
     - bcopy(KER2) .................................................... 36
     - biodone(KER2) ................................................... 38
     - biowait(KER2) ................................................... 40
     - brelse(KER2) ..................................................... 42
     - buf(KER4) ........................................................ 43
     - busywait(KER2) .................................................. 49
     - bzero(KER2) ...................................................... 50
     - copyin(KER2) ..................................................... 51
     - copyout(KER2) ................................................... 53
     - cspinlock(KER2) ................................................. 55
     - dealloc_spinlock(KER2) ....................................... 58
     - FREE(KER2) ....................................................... 59
     - get_sleep_lock(KER2) ......................................... 61
     - getc(KER2) ....................................................... 63
     - getcb(KER2) ..................................................... 65
     - getcf(KER2) ..................................................... 67
     - geteblk(KER2) .................................................. 68
     - gterror(KER2) .................................................... 70
     - gsignal(KER2) ................................................... 72
     - iovec(KER4) ...................................................... 74
     - kernel_iomap(KER2) .............................................. 75
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel_iomap_public(KER2)</td>
<td>78</td>
</tr>
<tr>
<td>kernel_iounmap(KER2)</td>
<td>81</td>
</tr>
<tr>
<td>kfree(KER2)</td>
<td>82</td>
</tr>
<tr>
<td>kmalloc(KER2)</td>
<td>83</td>
</tr>
<tr>
<td>Ktimeout(KER2)</td>
<td>85</td>
</tr>
<tr>
<td>ldsid(KER2)</td>
<td>87</td>
</tr>
<tr>
<td>major(KER2)</td>
<td>89</td>
</tr>
<tr>
<td>makedev(KER2)</td>
<td>90</td>
</tr>
<tr>
<td>MALLOC(KER2)</td>
<td>91</td>
</tr>
<tr>
<td>map_mem_to_host(KER2)</td>
<td>94</td>
</tr>
<tr>
<td>minor(KER2)</td>
<td>96</td>
</tr>
<tr>
<td>minphys(KER2)</td>
<td>97</td>
</tr>
<tr>
<td>msg_printf(KER2)</td>
<td>99</td>
</tr>
<tr>
<td>owns_spinlock(KER2)</td>
<td>101</td>
</tr>
<tr>
<td>p_pgrp(KER2)</td>
<td>103</td>
</tr>
<tr>
<td>panic(KER2)</td>
<td>105</td>
</tr>
<tr>
<td>physio(KER2)</td>
<td>106</td>
</tr>
<tr>
<td>printf(KER2)</td>
<td>110</td>
</tr>
<tr>
<td>privlbcopy(KER2)</td>
<td>112</td>
</tr>
<tr>
<td>psignal(KER2)</td>
<td>114</td>
</tr>
<tr>
<td>putc(KER2)</td>
<td>116</td>
</tr>
<tr>
<td>putcb(KER2)</td>
<td>118</td>
</tr>
<tr>
<td>putcf(KER2)</td>
<td>119</td>
</tr>
<tr>
<td>seltrue(KER2)</td>
<td>121</td>
</tr>
<tr>
<td>selwakeup(KER2)</td>
<td>123</td>
</tr>
<tr>
<td>sleep(KER2)</td>
<td>125</td>
</tr>
<tr>
<td>spinlock(KER2)</td>
<td>128</td>
</tr>
<tr>
<td>spinunlock(KER2)</td>
<td>130</td>
</tr>
<tr>
<td>sprintf(KER2)</td>
<td>132</td>
</tr>
<tr>
<td>strcmp(KER2)</td>
<td>134</td>
</tr>
<tr>
<td>strcat(KER2)</td>
<td>135</td>
</tr>
<tr>
<td>strcpy(KER2)</td>
<td>137</td>
</tr>
<tr>
<td>strlen(KER2)</td>
<td>139</td>
</tr>
<tr>
<td>strncpy(KER2)</td>
<td>141</td>
</tr>
<tr>
<td>strncmp(KER2)</td>
<td>143</td>
</tr>
<tr>
<td>suser(KER2)</td>
<td>145</td>
</tr>
<tr>
<td>sw_trigger(KER2)</td>
<td>146</td>
</tr>
<tr>
<td>timeout(KER2)</td>
<td>149</td>
</tr>
<tr>
<td>uiot(KER4)</td>
<td>151</td>
</tr>
<tr>
<td>uiomove(KER2)</td>
<td>153</td>
</tr>
<tr>
<td>unmmap_mem_from_host(KER2)</td>
<td>156</td>
</tr>
<tr>
<td>unttimeout(KER2)</td>
<td>157</td>
</tr>
</tbody>
</table>
user_iomap(KER2) ......................................................158
user_iounmap(KER2) ...................................................161
VASSERT(KER2) ............................................................162
waiting_in_select(KER2) ..............................................163
wakeup(KER2) ..............................................................165

3. CDIO Reference Pages
   Functions, Structures and Macro Commands ..........................169
dma_sync(CDIO3) ...........................................................170
dma_sync_IO(CDIO3) .......................................................173
drv_info(CDIO4) .............................................................176
drv_ops(CDIO4) .............................................................179
install_driver(CDIO3) .....................................................183

4. WSIO Reference Pages
   Functions, Macros and Structures ......................................187
bp_dma_cleanup(WSIO3) ................................................188
dma_clean(CDIO3) .........................................................190
dma_cleanup(WSIO3) .......................................................192
dma_parms(CDIO4) ........................................................193
dma_setup(WSIO3) ........................................................195
driver_addr_probe(WSIO_DRV) ........................................197
driver_attach(WSIO_DRV) ................................................199
driver_close(WSIO_DRV) ................................................201
driver_dev_init(WSIO_DRV) ............................................203
driver_dev_probe(WSIO_DRV) .........................................204
driver_if_init(WSIO_DRV) ................................................206
driver_install(WSIO_DRV) ..............................................207
driver_ioctl(WSIO_DRV) .................................................209
driver_isr(WSIO_DRV) ....................................................211
driver_minor_build(WSIO_DRV) ....................................212
driver_minphys(WSIO_DRV) ...........................................214
driver_open(WSIO_DRV) ...............................................216
driver_psize(WSIO_DRV) ...............................................218
driver_read(WSIO_DRV) ................................................219
driver_select(WSIO_DRV) .............................................221
driver_strategy(WSIO_DRV) .........................................223
driver_write(WSIO_DRV) ..............................................224
free_isc(WSIO3) ..........................................................226
get_new_isc(WSIO3) .....................................................227
iodone(WSIO3) ............................................................229
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iowait(WSIO3)</td>
<td>231</td>
</tr>
<tr>
<td>isc_claim(WSIO3)</td>
<td>233</td>
</tr>
<tr>
<td>isc_table_type(WSIO4)</td>
<td>235</td>
</tr>
<tr>
<td>m_wsio_funcnum(WSIO3)</td>
<td>239</td>
</tr>
<tr>
<td>m_wsio_selcode(WSIO3)</td>
<td>240</td>
</tr>
<tr>
<td>m_wsio_vsc(WSIO3)</td>
<td>241</td>
</tr>
<tr>
<td>mod_wsio_attach_list_add(WSIO3)</td>
<td>242</td>
</tr>
<tr>
<td>mod_wsio_attach_list_remove(WSIO3)</td>
<td>244</td>
</tr>
<tr>
<td>wsio_activate_probe(WSIO3)</td>
<td>246</td>
</tr>
<tr>
<td>wsio_alloc_mem(WSIO3)</td>
<td>249</td>
</tr>
<tr>
<td>wsio_alloc_mem_handle(WSIO3)</td>
<td>251</td>
</tr>
<tr>
<td>wsio_allocate_dma_handle(WSIO3)</td>
<td>254</td>
</tr>
<tr>
<td>wsio_allocate_shared_mem(WSIO3)</td>
<td>256</td>
</tr>
<tr>
<td>WSIO_BIG_ENDIAN(WSIO3)</td>
<td>260</td>
</tr>
<tr>
<td>wsio_cfg_inXX(WSIO3)</td>
<td>262</td>
</tr>
<tr>
<td>wsio_cfg_outXX(WSIO3)</td>
<td>264</td>
</tr>
<tr>
<td>wsio_dma_pass_thru(WSIO3)</td>
<td>266</td>
</tr>
<tr>
<td>wsio_dma_set_device_attributes(WSIO3)</td>
<td>269</td>
</tr>
<tr>
<td>wsio_drv_data_t(WSIO4)</td>
<td>274</td>
</tr>
<tr>
<td>wsio_drv_info(WSIO4)</td>
<td>276</td>
</tr>
<tr>
<td>wsio_event_t(WSIO5)</td>
<td>277</td>
</tr>
<tr>
<td>wsio_fastmap_dma_buffer(WSIO3)</td>
<td>279</td>
</tr>
<tr>
<td>wsio_flush_shared_mem(WSIO3)</td>
<td>282</td>
</tr>
<tr>
<td>wsio_free_dma_handle(WSIO3)</td>
<td>284</td>
</tr>
<tr>
<td>wsio_free_mem(WSIO3)</td>
<td>286</td>
</tr>
<tr>
<td>wsio_free_mem_handle(WSIO3)</td>
<td>287</td>
</tr>
<tr>
<td>wsio_free_shared_mem(WSIO3)</td>
<td>288</td>
</tr>
<tr>
<td>wsio_generic_event_t(WSIO5)</td>
<td>290</td>
</tr>
<tr>
<td>wsio_get_active_processor_count(WSIO3)</td>
<td>292</td>
</tr>
<tr>
<td>wsio_get_all_registers(WSIO3)</td>
<td>293</td>
</tr>
<tr>
<td>wsio_get_iports(WSIO3)</td>
<td>295</td>
</tr>
<tr>
<td>wsio_get_isc(WSIO3)</td>
<td>297</td>
</tr>
<tr>
<td>wsio_get_processor_count(WSIO3)</td>
<td>299</td>
</tr>
<tr>
<td>wsio_get_system_params(WSIO3)</td>
<td>300</td>
</tr>
<tr>
<td>wsio_init_map_context(WSIO3)</td>
<td>302</td>
</tr>
<tr>
<td>wsio_install_driver(WSIO3)</td>
<td>304</td>
</tr>
<tr>
<td>wsio_install_drv_event_handler(WSIO3)</td>
<td>307</td>
</tr>
<tr>
<td>wsio_intr_activate(WSIO5)</td>
<td>309</td>
</tr>
<tr>
<td>wsio_intr_alloc(WSIO3)</td>
<td>312</td>
</tr>
<tr>
<td>wsio_intr_deactivate(WSIO5)</td>
<td>314</td>
</tr>
<tr>
<td>wsio_intr_deactivate_nowait(WSIO3)</td>
<td>317</td>
</tr>
<tr>
<td>wsio_intr_free(WSIO3)</td>
<td>320</td>
</tr>
</tbody>
</table>
5. Network Device Driver Reference Pages

Network Macros, Structures, and Functions ........................................ 409
ETHER_MAP_IP_MULTICAST .................................................. 410
hw_ift(NET4) ............................................................ 411
6. PCI Reference Pages

PCI Macros and Functions

CONNECT_INIT_ROUTINE(PCI3) ...................................................... 472
PCI_ATTACH_DEV_INIT_ERROR(PCI3) ............................................. 473
pci_desc_bus_transactions_isc(PCI3) ............................................ 474
pci_get_fru_info_isc(PCI3) ....................................................... 475
pci_get_port_hndl_isc(PCI3) ....................................................... 476
pci_read_cfg_uintN_isc(PCI3) ...................................................... 477
pci_read_port_uintN_isc(PCI3) ..................................................... 478
pci_unget_port_hndl_isc(PCI3) .................................................... 479
pci_write_cfg_uintN_isc(PCI3) ..................................................... 480
pci_write_port_uintN_isc(PCI3) .................................................... 481
READ_REG_UINTn_ISC(PCI3) ......................................................... 482
WRITE_REG_UINTn_ISC(PCI3) ......................................................... 483
PCI_ERRATA-1(PCI5) ................................................................. 484
PCI_ERRATA-2(PCI5) ................................................................. 485
PCI_ERRATA-3(PCI5) ................................................................. 486
### 7. SCSI Reference Pages

**SCSI Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd_close(SCSI_DRV)</td>
<td>506</td>
</tr>
<tr>
<td>dd_done(SCSI_DRV)</td>
<td>507</td>
</tr>
<tr>
<td>dd_ioctl(SCSI_DRV)</td>
<td>508</td>
</tr>
<tr>
<td>dd_ioctl_okay(SCSI_DRV)</td>
<td>509</td>
</tr>
<tr>
<td>dd_open(SCSI_DRV)</td>
<td>510</td>
</tr>
<tr>
<td>dd_pass_thru_done(SCSI_DRV)</td>
<td>511</td>
</tr>
<tr>
<td>dd_pass_thru_okay(SCSI_DRV)</td>
<td>512</td>
</tr>
<tr>
<td>dd_read(SCSI_DRV)</td>
<td>513</td>
</tr>
<tr>
<td>dd_start(SCSI_DRV)</td>
<td>514</td>
</tr>
<tr>
<td>dd_strategy(SCSI_DRV)</td>
<td>515</td>
</tr>
<tr>
<td>dd_write(SCSI3)</td>
<td>516</td>
</tr>
<tr>
<td>driver_if_abort(SCSI_DRV2)</td>
<td>517</td>
</tr>
<tr>
<td>driver_if_bdr(SCSI_DRV2)</td>
<td>518</td>
</tr>
<tr>
<td>driver_if_close(SCSI_DRV2)</td>
<td>519</td>
</tr>
<tr>
<td>driver_if_open(SCSI3)</td>
<td>520</td>
</tr>
<tr>
<td>driver_if_reset_bus(SCSI_DRV2)</td>
<td>521</td>
</tr>
<tr>
<td>driver_if_start(SCSI_DRV2)</td>
<td>523</td>
</tr>
<tr>
<td>m_bus_id(SCSI3)</td>
<td>525</td>
</tr>
<tr>
<td>m_lun_id(SCSI3)</td>
<td>527</td>
</tr>
<tr>
<td>m_tgt_id(SCSI3)</td>
<td>529</td>
</tr>
<tr>
<td>m_scsi_bus(SCSI3)</td>
<td>531</td>
</tr>
<tr>
<td>m_scsi_isct(SCSI3)</td>
<td>533</td>
</tr>
<tr>
<td>m_scsi_lun(SCSI3)</td>
<td>535</td>
</tr>
<tr>
<td>m_scsi_tgt(SCSI3)</td>
<td>537</td>
</tr>
<tr>
<td>scb(SCSI3)</td>
<td>539</td>
</tr>
<tr>
<td>scsi_action(SCSI3)</td>
<td>545</td>
</tr>
<tr>
<td>scsi_bus_lock(SCSI3)</td>
<td>548</td>
</tr>
<tr>
<td>scsi_bus_unlock(SCSI3)</td>
<td>550</td>
</tr>
<tr>
<td>scsi_cbfm(SCSI3)</td>
<td>552</td>
</tr>
<tr>
<td>scsi_cmd(SCSI3)</td>
<td>553</td>
</tr>
<tr>
<td>scsi_cmdx(SCSI3)</td>
<td>555</td>
</tr>
<tr>
<td>scsi_ddsw(SCSI4)</td>
<td>557</td>
</tr>
<tr>
<td>scsi_dequeue(SCSI3)</td>
<td>561</td>
</tr>
<tr>
<td>scsi_dequeue_bp(SCSI3)</td>
<td>563</td>
</tr>
<tr>
<td>scsi_enqueue(SCSI3)</td>
<td>565</td>
</tr>
<tr>
<td>scsi_ifsw(SCSI3)</td>
<td>567</td>
</tr>
<tr>
<td>scsi_init_inquiry_data(SCSI3)</td>
<td>571</td>
</tr>
<tr>
<td>scsi_ioctl(SCSI3)</td>
<td>572</td>
</tr>
<tr>
<td>scsi_lun_close(SCSI3)</td>
<td>585</td>
</tr>
<tr>
<td>scsi_lun_open(SCSI3)</td>
<td>587</td>
</tr>
</tbody>
</table>
A. Deprecated WSIO Interfaces

Functions

init_map_context(CDIO3) .................................................. 598
isrlk(WSIO3) ................................................................. 600
isrunlink(WSIO3) ........................................................... 602
m_instance(WSIO3) ......................................................... 604
wsio_allocate_shared_memory(WSIO3) .............................. 605
wsio_fastmap(WSIO3) ....................................................... 608
wsio_flush_shared_memory(WSIO3) .................................... 611
wsio_free_shared_memory(WSIO3) ....................................... 613
wsio_get_interrups(WSIO3) ............................................... 615
wsio_get_pva(WSIO3) ....................................................... 617
wsio_get_registers(WSIO3) ............................................... 620
wsio_install_drv_func(WSIO_DRV) ..................................... 622
wsio_map(WSIO3) ............................................................ 625
wsio_register_probe_func(WSIO3) ...................................... 634
wsio_remap(WSIO3) ........................................................ 636
wsio_set_attributes(WSIO3) ............................................... 638
wsio_unmap(WSIO3) ........................................................ 640
1 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 `man` page 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
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

#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

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);
b_cpsema (KER2)

/*! 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

\texttt{b\_initsema(KER2)} – Initialize a beta semaphore.

SYNOPSIS

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

void b_initsema (struct b_sema *sema, int val, int order, char *name);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \texttt{sema} \hspace{1cm} Pointer to a \texttt{b\_sema} structure.
  \item \texttt{val} \hspace{1cm} Initial value of \texttt{sema}. Normally set to 1.
  \item \texttt{order} \hspace{1cm} Lock order.
  \item \texttt{name} \hspace{1cm} Beta semaphore name.
\end{itemize}

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 \texttt{b\_initsema()} kernel function initializes the beta semaphore pointed to by the \texttt{sema} parameter. The caller is responsible for allocating the kernel memory that instantiates the \texttt{b\_sema} structure.

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

The \texttt{order} parameter is the lock order of the beta semaphore. \texttt{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 <\texttt{sys/sem\_global.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

/*
 * 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

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.
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

\texttt{b\_psema} (KER2) – Acquire (lock) a beta semaphore.

SYNOPSIS

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

void b_psema (struct b_sema * sema);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \texttt{sema} \hspace{1cm} Pointer to a \texttt{b\_sema} structure.
\end{itemize}

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 \texttt{b\_psema()} kernel function attempts to acquire (lock) a beta semaphore pointed to by \texttt{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

```c
static b_sema_t mydrv_sema;
...

/*
 * Acquire the beta semaphore for my driver.
 */
bl_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.
 */
bl_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>

void 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

```c
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 (void * s1, void * s2, size_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

- `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

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

void bcopy (void *from, void *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 `uio_move()`.

To copy data between buffers in other user spaces, use `privlbcopy()`.

RETURN VALUES

None

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_iiodone. 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

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 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 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 with future 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>int32_t</td>
<td>b_blkno</td>
</tr>
</tbody>
</table>
Table 2-1  Relevant buf Structure Fields (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>daddr_t</td>
<td>b_blkno</td>
</tr>
<tr>
<td>int32_t</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>int32_t</td>
<td>b_flags</td>
</tr>
<tr>
<td>int (*)()</td>
<td>b_iiodone</td>
</tr>
<tr>
<td>struct buf *</td>
<td>b_merge</td>
</tr>
<tr>
<td>uint16_t</td>
<td>b_merge_cnt</td>
</tr>
<tr>
<td>unsigned int</td>
<td>b_resid</td>
</tr>
<tr>
<td>intptr_t</td>
<td>b_s2</td>
</tr>
<tr>
<td>char</td>
<td>b_s3</td>
</tr>
<tr>
<td>intptr_t</td>
<td>b_s7</td>
</tr>
<tr>
<td>intptr_t</td>
<td>b_s8</td>
</tr>
<tr>
<td>space_t</td>
<td>b_spaddr</td>
</tr>
<tr>
<td>caddr_t</td>
<td>b_un.b_addr</td>
</tr>
<tr>
<td>uint16_t</td>
<td>b2_flags</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**  
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 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>B_ASYNC</td>
<td>Buffer write is synchronous. Do not wait for I/O completion. Mutually exclusive with <code>B_SYNC</code>.</td>
</tr>
<tr>
<td>B_BCACHE</td>
<td>The buffer is allocated from the file system buffer cache.</td>
</tr>
<tr>
<td>B_BUSY</td>
<td>The buffer is in use.</td>
</tr>
<tr>
<td>B_CACHE</td>
<td><code>bread()</code> located this buffer in the cache.</td>
</tr>
<tr>
<td>B_CALL</td>
<td><code>iodone()</code> is to call the function pointed to by <code>b_iiodone</code>.</td>
</tr>
<tr>
<td>B_DELWRI</td>
<td>Delayed write. Write at exit of <code>avail</code> list processing by the buffer cache management code.</td>
</tr>
<tr>
<td>B_DONE</td>
<td>The buffer transfer has completed. <code>biodone()</code> sets this flag.</td>
</tr>
<tr>
<td>B_END_OF_DATA</td>
<td>This flag is used to terminate, without error, a <code>physio</code> transfer, with less than <code>b_count</code> bytes transferred.</td>
</tr>
<tr>
<td>B_ERROR</td>
<td>An error occurred during the I/O transfer. If the driver sets this flag, it must also set the <code>b_error</code> field with an <code>errno</code> value.</td>
</tr>
<tr>
<td>B_FSYSIO</td>
<td>Buffer came from <code>bread()</code> or <code>bwrite()</code>.</td>
</tr>
<tr>
<td>B_INVAL</td>
<td>The buffer does not contain valid information.</td>
</tr>
</tbody>
</table>
**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.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_WRITE</td>
<td>A pseudo flag that semantically indicates &quot;not B_READ&quot;. 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:</td>
</tr>
<tr>
<td></td>
<td>if (!(bp-&gt;b_flags &amp; B_READ) )</td>
</tr>
<tr>
<td></td>
<td>The expression (bp-&gt;b_flags &amp; B_WRITE) is always zero.</td>
</tr>
<tr>
<td>B_WRITEV</td>
<td>This flag is used by LVM when attempting to correct disk soft errors, and should not be touched by a driver.</td>
</tr>
<tr>
<td>b_iiodone</td>
<td>Pointer to a function that iiodone() 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.</td>
</tr>
<tr>
<td>b_merge</td>
<td>Pointer to the next buf structure where the list of buffers have been logically merged together. Valid only when B2_LIST is set in b2_flags.</td>
</tr>
<tr>
<td>b_merge_cnt</td>
<td>Number of buffers merged together through b_merge.</td>
</tr>
<tr>
<td>b_resid</td>
<td>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().</td>
</tr>
<tr>
<td>b_s2</td>
<td>Scratch field for driver use. For example, SCSI Interface Drivers use this field to store a pointer to a SCSI Control Block.</td>
</tr>
<tr>
<td>b_s3</td>
<td>Scratch field for driver use. For example, the SCSI Subsystem uses this field to store state information.</td>
</tr>
<tr>
<td>b_s7</td>
<td>Scratch field for driver use. For example, the SCSI Subsystem reserves this field for device drivers.</td>
</tr>
<tr>
<td>b_s8</td>
<td>Scratch field for driver use. For example, the SCSI subsystem reserves this field for device drivers.</td>
</tr>
<tr>
<td>b_spaddr</td>
<td>Space ID of the buffer specified by the buffer header. Do not assume this value to be KERNELSPACE.</td>
</tr>
</tbody>
</table>
**buf(KER4)**

*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().

*b2_flags* Information about the buffer in addition to *b_flags*. This value is composed from the following bit flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>If set...</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2_LIST</td>
<td>Buffer is linked with other buffers through the <em>b_merge</em> field.</td>
</tr>
</tbody>
</table>

**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

```
#include <sys/kern_svc.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>

void bzero (void * 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

#include <sys/kern_svcs.h>

int copyin (void * from_user, void * 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

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 (void * from_kernel, void * 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

- `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_buff))) {
    return EFAULT;
}
```

SEE ALSO

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

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

SYNOPSIS

#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

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

CONSTRAINTS
EXAMPLE

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(KER2), cspinlock(KER2), get_sleep_lock(KER2), owns_spinlock(KER2), spinlock(KER2), spinunlock(KER2)
NAME
FREE(KER2) – 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

```c
struct buf * bp;

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

/*
 * 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

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

lock_t * get_sleep_lock (void * 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:

```c
(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()`.
Kernel Reference Pages
get_sleep_lock(KER2)

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

crtc(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

 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)

    /* concatenate characters from one clist to another */

    {
        int c;

        while ((c = getc(from)) >= 0)
            (void) putc(c, to);
    }
Kernel Reference Pages

getc(KER2)

**SEE ALSO**

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

getcb(KER2) – Manipulate cblocks on a clist

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

<>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);

    ...
}
SEE ALSO

putcb(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

<>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

```c
struct buf * bp;

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

SEE ALSO

`brelse(KER2), buf(KER4)`
NAME

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

SYNOPSIS

#include <sys/buf.h>

int gterror (struct buf * bp);

PARAMETERS

bp  Pointer to a buf structure.

DESCRIPTION

gterror() 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

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

CONSTRAINTS
EXAMPLE

```c
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

#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 2-2 The iovec Structure Fields

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

uioc(KER4), wsio_fastmap(WSIO3), wsio_map(WSIO3), wsio_remap(WSIO3)
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 phys_addr
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

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

NULL Error.
EXAMPLES

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

#include <sys/iomap.h>
#include <sys/mman.h>
#include <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 */
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.
*/
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

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

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

PARAMETERS

- `virt_addr`: NULL or equivalent to `phys_addr`
- `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

- `<NULL`: Successful completion. The value is the host virtual address for accessing the space specified by the parameters.
- `NULL`: Error.
EXAMPLES

/*
** example maps 1 page of I/O space at
** 0xf0000000 with no protection
*/
#include <sys/iomap.h>
#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 unmmap `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

#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

   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 (void *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>

void * kmalloc (size_t size, int type, arena_flags_t 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. Independent Hardware Vendors should specify the M_IHV pool for general I/O purposes of their drivers.

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**

- `<>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

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

callout_t * 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 tountimeout().
**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

#include <sys/kern_svcs.h>

space_t ldsid (void * 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

/ *
 * 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));
Kernel Reference Pages
Idsid(KER2)

SEE ALSO

privlbcopy(KER2)
NAME

**major**(KER2) – Extract the major number from a device number

SYNOPSIS

```c
#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 & 0xffffff)))

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:

\begin{itemize}
\item called in an interrupt context, or
\item called while holding a spinlock
\end{itemize}

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 \texttt{flags} parameter if the caller is able to block and wait for memory to become available.

\section*{RETURN VALUES}

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

\begin{itemize}
\item <\texttt{NULL}> Virtual address of kernel memory allocated.
\item \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.
\end{itemize}

\section*{CONSTRAINTS}

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

\begin{itemize}
\item the caller must \textit{not} be in an interrupt context, and
\item no spinlocks can be held
\end{itemize}

\section*{EXAMPLES}

\begin{verbatim}
my_struct_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.
     */
}
\end{verbatim}
SEE ALSO

FREE(KER2), kfree(KER2), kmalloc(KER2)
NAME
map_mem_to_host(KER2) – Map physical bus address to host virtual space

SYNOPSIS

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

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

PARAMETERS

- `isc` : ISC 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 bit 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

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

CONSTRAINTS
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)&0xffffff))
```

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
Kernel Reference Pages

minphys(KER2)

**SEE ALSO**

physio(KER2), driver_minphys(WSIO_DRV)
NAME

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

SYNOPSIS

#include <sys/kern_svcs.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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Printed Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>%; no argument</td>
</tr>
<tr>
<td>%b</td>
<td>Characters from string argument; can include backslash-escape codes</td>
</tr>
<tr>
<td>%c</td>
<td>Character from integer argument</td>
</tr>
<tr>
<td>%d</td>
<td>Signed decimal from integer argument</td>
</tr>
<tr>
<td>%lx</td>
<td>The argument x is long integer; x is one of x, d, u, or o</td>
</tr>
<tr>
<td>%o</td>
<td>Octal from integer argument</td>
</tr>
<tr>
<td>%s</td>
<td>Characters from string argument</td>
</tr>
<tr>
<td>%u</td>
<td>Unsigned decimal from integer argument</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal from integer argument</td>
</tr>
</tbody>
</table>
These formats are the same as in \textit{print} (3S). Other formats specified in \textit{printf} (3S) are not supported.

**RETURN VALUES**

\texttt{msg\_printf()} returns the length of the formatted string.

**CONSTRAINTS**

**SEE ALSO**

\texttt{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

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

#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

#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);
Kernel Reference Pages
p_pgrp(KER2)

**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

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

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

PARAMETERS

- `strat` : 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 `strat` function, and waits for the I/O to complete.

The `strat` parameter is a pointer to the driver strategy function. `physio()` calls `strat` to start the I/O transfer, then waits by calling `biowait()`. When the I/O transfer completes, `strat` calls `biodone()` to awaken the waiting thread. `strat` 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 *uioc* structure that is passed to the driver. The *uioc* structure specifies the following:

- **uiociov** Pointer to an *iovec* structure that contains the base address *iov_base* and transfer length *iov_len* of the I/O request.
- **uiociovct** Number of *iovec* structures. If >1, *uiociov* points to an array of *iovec* structures.
- **uiocoff** Offset into device.
- **uiocseg** Type of memory segment to transfer. If set to *UIOSEG_USER*, *physio()* must be called in the user context.
- **uiocresid** Number of bytes of data remaining to be transferred.

For each *iovec* structure, *physio()* performs the following:

1. Validate the user has appropriate access permissions for the data pages specified by *iovec*.
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 *uiocoff*. 
3. Call \texttt{mincnt} to adjust the transfer length, if too large. If adjusted, \texttt{physio()} will make multiple calls to \texttt{strat} until all the data specified by \texttt{iov_len} has been transformed (or an error occurs).

4. If \texttt{uio\_seg} is not \texttt{UIOSEG\_KERNEL}, lock down the data pages to be transferred.

5. If \texttt{uio\_seg} is not \texttt{UIOSEG\_KERNEL} and the driver has set \texttt{C\_MAP\_BUFFER\_TO\_KERNEL} in the \texttt{d\_flags} field of its \texttt{drv\_ops\_t} structure, \texttt{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 \texttt{strat} passing the \texttt{buf} structure pointer \texttt{bp} as a parameter, then wait for the I/O request to complete by calling \texttt{biowait()}.

7. The driver calls \texttt{biodone()} when the I/O request completes to awaken the thread waiting in \texttt{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:
   \begin{itemize}
   \item \texttt{uio\_resid} is decreased by the transfer length
   \item \texttt{uio\_offset} is increased by the transfer length
   \end{itemize}

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

**RETURN VALUES**

\begin{itemize}
\item 0 Successful completion.
\item <>0 Error.
\end{itemize}

**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,
                  minphys, uio);
}
```

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Printed Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>%; no argument</td>
</tr>
</tbody>
</table>
| %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 |
Kernel Reference Pages
printf(KER2)

%xHexadecimal 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)
NAME

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

SYNOPSIS

#include <sys/kern_svcs.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 ldsid() 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_MAPBUFFER_TO_KERNEL in its drv_ops_t structure.)

RETURN VALUES

0 Successful completion.
<0 Error.

CONSTRAINTS

EXAMPLES

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

WARNINGS

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 panic().

SEE ALSO

bcopy(KER2), copyin(KER2), copyout(KER2), ldsid(KER2)
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

```c
#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

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

CONSTRAINTS
EXAMPLES

```c
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))
            break;
}
```

SEE ALSO

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

putcb(KER2) – Manipulate cblocks on a clist

SYNOPSIS

#include <sys/clist.h>

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

PARAMETERS

  cb Pointer to a cblock.
  list Pointer to a clist.

DESCRIPTION

putcb() adds the cblock pointed to by cb onto the clist 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);
    ...
}
Kernel Reference Pages
putc(KER2)

**SEE ALSO**

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

seltrue(KER2) – Select driver entry point that returns true for traditional select flags.

SYNOPSIS

int seltrue (dev_t dev, int flag);

PARAMETERS

dev
Device number.
flag
Device select flags.

DESCRIPTION

The seltrue() kernel function may be used as the driver entry point for the select() system call. It returns true for the traditional select flag values of FREAD, FWRITE, and 0.

RETURN VALUES

seltrue() returns the following values:

0       Device select flag is not a traditional select option.
-1      Device select flag is either FREAD, FWRITE, or 0.

CONSTRAINTS
EXAMPLES

```c
drv_ops_t lpr0_drv_ops = {       /* driver entry points */
    lpr0_open,                /* open */
    lpr0_close,               /* close */
    NULL,                     /* strategy */
    NULL,                     /* dump */
    NULL,                     /* psize */
    NULL,                     /* reserved */
    lpr0_read,                /* read */
    lpr0_write,               /* write */
    lpr0_ioctl,               /* ioctl */
    seltrue,                  /* select */
    NULL,                     /* option1 */
    NULL, NULL, NULL, NULL,   /* reserved entry points */
    0,                        /* device flags */
};
```

SEE ALSO

select(2)
NAME

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

SYNOPSIS

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

PARAMETERS

threadp 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 threadp 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 (kthreadp & waiting_in_select (kthreadp) {
        * There is going to be a collision of multiple
        * threads sleeping on a select condition.
        */
        myselstruct->selflag |= MYSEL_COLLISION;
    } else {
        /*
        * Save the kernel thread pointer of this
        * thread that will sleep on a select condition.
        */
        myselstruct->thread = u.u_kthreadp;
    }
    ...
}

static void
mydev_selwakeup(void)
{
    selwakeup(myselstruct->thread,
              myselstruct->selflag & MYSEL_COLLISION);
    myselstruct->thread = NULL;
    myselstruct->selflag &= ~MYSEL_COLLISION;
}

SEE ALSO

waiting_in_select (KER2)
NAME

sleep(KER2) – Sleep on a channel

SYNOPSIS

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

int sleep (void * 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**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Awakened by a corresponding call to <code>wakeup()</code></td>
</tr>
<tr>
<td>1</td>
<td>Awakened by a signal if the <code>PCATCH</code> flag is set</td>
</tr>
</tbody>
</table>

**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),
geet_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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Printed Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>%; no argument</td>
</tr>
<tr>
<td>%b</td>
<td>Characters from string argument; can include backslash-escape codes</td>
</tr>
<tr>
<td>%c</td>
<td>Character from integer argument</td>
</tr>
<tr>
<td>%d</td>
<td>Signed decimal from integer argument</td>
</tr>
<tr>
<td>%lx</td>
<td>The argument ( x ) is long integer; ( x ) is one of ( x, d, u, ) or ( o ).</td>
</tr>
<tr>
<td>%o</td>
<td>Octal from integer argument</td>
</tr>
</tbody>
</table>
%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_svcs.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 char.

RETURN VALUES

strcat() returns buf.

CONSTRAINTS

SEE ALSO

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

strcmp(KER2) – Compare two strings

SYNOPSIS

#include <sys/kern_svcs.h>

int strcmp (const char * s1, const 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

0 The strings are identical.

<>0 The strings are different.

CONSTRAINTS
EXAMPLES

```c
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, const char *s2);

PARAMETERS

s1 Pointer to the destination string.

s2 Pointer to the source 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);
  }
  
}

SEE ALSO

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

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

SYNOPSIS

```c
#include <sys/kern_svcs.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++)
    {
        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

strncmp(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

- 0 The strings are equivalent.
- \(<0\) The strings are different.

CONSTRAINTS
EXAMPLES

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

#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

{
   .
   .
   /*
   ** Get the first four bytes of the inquiry data
   ** for dev_id and the vendor and product id for desc
   */
   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
suser(KER2) – Test if the current user is a superuser

SYNOPSIS

#include <sys/kern_svcs.h>

int suser (void);

PARAMETERS
None.

DESCRIPTION
The suser() kernel function tests if the current user is a superuser.

RETURN VALUES

0 The current user is a superuser.
-1 The current user is not a superuser.

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 *),
                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

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

callout_t * 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.
KERNEL REFERENCE PAGES

timeout(KER2)

CONSTRAINTS

Must not be called while holding a spinlock of order >= 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), untimeout(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>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>
<tr>
<td>off_t</td>
<td>uio_offset</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 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.

**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.

**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

#include <sys/uio.h>

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

PARAMETERS

addr Address of kernel memory.
n Number of bytes to copy
flag 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 flag 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. If `uio_seg` is `UIO_SYSSPACE`, kernel space is to be accessed and `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**

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

**CONSTRAINTS**

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.

If `uio_seg` is `UIO_SYSSPACE`, kernel space is to be accessed and `uiomove()` can be called in any user context or in an interrupt context.
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, uiop);
    brelse(bp);
    ...
}
```

SEE ALSO

brelse(KER2), geteblk(KER2)
unmap_mem_from_host(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
  ≥ 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 phys_addr
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

<>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 ** 0xff0000000 to driver and the currently ** running process with user read/write ** protection */
#include <sys/iomap.h>
#include <sys/mman.h>
.
.
{
    int sm;
    char byte;
    caddr_t mapped_addr;
    caddr_t user_mapped_addr;
    caddr_t phys_addr = 0xff0000000;
    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, u_int 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

1 Error.

0 Success.

CONSTRAINTS

SEE ALSO

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

**VASSERT(KER2)** – Test an assertion if OSDEBUG kernel

SYNOPSIS

```c
#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

waiting_in_select (KER2) – Determines if a kernel thread is waiting on a select condition.

SYNOPSIS

int waiting_in_select (struct kthread * threadp);

PARAMETERS

threadp Pointer to a kernel thread.

DESCRIPTION

The waiting_in_select() kernel function determines if the specified kernel thread threadp is waiting on a select condition. This function replaces accesses to the global variable selwait which is no longer supported.

RETURN VALUES

waiting_in_select() returns the following values:

0 Thread is not waiting on a select condition.
1 Thread is waiting on a select condition.

CONSTRAINTS
EXAMPLES

```
#define MYSEL_COLLISION 1 /* my select collision flag */

static int mydev_select(dev_t dev, int rw)
{
    struct kthread * kthreadp;
    ...
    /*
    * Check if another kernel thread is already
    * sleeping on a select condition with the driver.
    */
    kthreadp = myselstruct->thread;
    if ((kthreadp &
         waiting_in_select(kthreadp)) {
        /*
        * There is going to be a collision of multiple
        * threads sleeping on a select condition.
        */
        myselstruct->selflag |= MYSEL_COLLISION;
    } else {
        /*
        * Save the kernel thread pointer of this
        * thread that will sleep on a select condition.
        */
        myselstruct->thread = u.u_kthreadp;
    }
    ...
}

static void mydev_selwakeup(void)
{
    selwakeup(myselstruct->thread,
              myselstruct->selflag & MYSEL_COLLISION);
    myselstruct->thread = NULL;
    myselstruct->selflag &= -MYSEL_COLLISION;
}
```

SEE ALSO

selwakeup(KER2)
NAME

\texttt{wakeup} (KER2) – Wake up all threads sleeping on a channel.

SYNOPSIS

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

int wakeup (void * chan);
\end{verbatim}

PARAMETERS

\begin{verbatim}
chan
\end{verbatim} Channel passed in the corresponding call to \texttt{sleep()}.

DESCRIPTION

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

The \texttt{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 \texttt{chan}. \texttt{chan} must not be zero.

If more than one thread sleeps on \texttt{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 \texttt{sleep()} should verify it is to handle the event; threads that do not handle the event may need to call \texttt{sleep()} again.

RETURN VALUES

\texttt{wakeup()} returns the number of threads awakened.

CONSTRAINTS

Must not be called while holding a spinlock of order greater than \texttt{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)
3 CDIO Reference Pages
This chapter contains reference pages for driver support routines that are external to all CDIOs.
Functions, Structures and Macro Commands
NAME

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

SYNOPSIS

#include <sys/dma.h>

#define dma_sync (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 Bit-wise 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 semicoherent platforms. The
declared hints are:

IO_ACCESSED 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 SYNCDMA
instructions.

IO_PREFETCHED Perform function only if the processor
prefetches data.

IO_READ Purge processor caches for inbound
data on noncoherent systems.

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).
DESCRIPTION

The dma_sync() CDIO macro has been superseded by the dma_sync_IO() function. New drivers are encouraged to call 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 `dma_sync()` with the `IO_READ` hint set. On noncurrent platforms, this will cause the associated processor caches to be purged of data that may have been perfected. 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.

**CONTAINTS**

**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

void dma_sync_IO (uint32_t addr_type, void * addr, int length,
                 uint32_t 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       Bit-wise 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 semicoherent platforms; nothing is done on
fully coherent platforms. The defined hints are:

IO_ACCESSED 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 SYNCDMA
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 SYNCDMA instruction on
semicoherent platforms. Done prior
to starting the DMA data transfer.
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 flushed to memory when used with `IO_WRITE` (even on coherent platforms).
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

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

PARAMETERS

```c
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 implementation of kernel functions that access name require that the string be less than 16 characters long.</td>
</tr>
<tr>
<td>class</td>
<td>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.</td>
</tr>
<tr>
<td>flags</td>
<td>The bit-wise OR of flag values that describe the driver, taken from:</td>
</tr>
<tr>
<td></td>
<td>DRV_CHAR Character device driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_BLOCK Block device driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_PSEUDO Pseudo driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_SCAN Driver supports bus scanning.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>b_major</td>
<td>The major number if this is a block device. Set it to -1 for dynamic assignment or if it is not a block device.</td>
</tr>
</tbody>
</table>
**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 (*d_select)(); /* 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 drv_ops_t CDIO structure type, defined in <sys/conf.h>, contains pointers to all driver entry points. A drv_ops_t structure must be statically allocated.

The relevant fields are described in the STRUCTURE MEMBERS section. All other fields in drv_ops_t should be NULL. Except as noted, entry points that don't apply to your driver or that your driver does not provide should be NULL (for example, 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 driver_open() routine, which enables a device for subsequent operations.</td>
</tr>
<tr>
<td>d_close()</td>
<td>both</td>
<td>Pointer to your driver_close() 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 driver_strategy() 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 driver_psize() 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 driver_read() 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 driver_write() routine, which should write the requested data to the device.</td>
</tr>
<tr>
<td>d_ioctl()</td>
<td>character</td>
<td>Pointer to your driver_ioctl() routine, which sends control information to, or gets it from, a device.</td>
</tr>
</tbody>
</table>
The flag bit defines for \texttt{d\_flags} are:

\texttt{C\_ALLCLOSES} \hspace{1em} Force a call to \texttt{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.)

---

**Table 3-1  Device Driver Fields in \texttt{drv\_ops\_t} Structure Type (Continued)**

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{d_select()}</td>
<td>character</td>
<td>Pointer to your \texttt{driver_select()} routine, which you can use to test for I/O completion and driver-dependent exception conditions. If your device is always ready for reading or writing, you can put \texttt{seltrue} in the \texttt{d_select()} field. If you do, calls to \texttt{select()} always return \texttt{true} without invoking your driver.</td>
</tr>
<tr>
<td>\texttt{pfilter}</td>
<td>both</td>
<td>Pointer to a \texttt{pfilter_t} structure. Use the &amp;\texttt{cpd_pfilter} pointer. This structure provides backward compatible routines for disk structures with fixed partitions, such as the Series 800 computers before the availability of the Logical Volume Manager (LVM). The &amp;\texttt{cpd_pfilter} pointer is required for such disks; it is ignored under other conditions (or you can use \texttt{NULL}).</td>
</tr>
<tr>
<td>\texttt{d_flags}</td>
<td>both</td>
<td>The bit-wise OR of flag values that indicate special features of the device. The flags give information about the device to the kernel. Drivers receive this information, but usually only validate it. Use 0 if no flags are set.</td>
</tr>
</tbody>
</table>
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

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

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)
4   WSIO Reference Pages
This chapter contains reference pages describing routines and data structures used by drivers to communicate with the WSIO CDIO. Some earlier interfaces have been deprecated and are now located in Appendix A.
Functions, Macros and Structures
NAME
bp_dma_cleanup(WSIO3) – Cleanup after a DMA transfer for a list of buffers.

SYNOPSIS

```c
void bp_dma_cleanup (struct isc_table_type *isc, struct buf *bp,
struct bp_dma_parms *bp_dma_parms);
```

PARAMETERS

- `isc` Pointer to an ISC structure.
- `bp` Pointer to the first buf structure
- `bp_dma_parms` Pointer to a bp_dma_parms structure.

DESCRIPTION

The `dma_cleanup()` WSIO function performs the required cleanup after a DMA transfer has completed for a list of buffers.

RETURN VALUES

None

CONSTRAINTS

EXAMPLES

```c
struct bp_dma_parms *bp_dma_parms = &lsp->bp_dma_parms;
struct dma_parms *dma_parms = &bp_dma_parms->dma_parms;

if (bp->b_merge_cnt == 0) {
    (void)dma_cleanup(isc, dma_parms);
} else {
    bp_dma_cleanup(isc, bp, bp_dma_parms);
}
```
SEE ALSO

bp_dma_setup(WSIO3), dma_cleanup(WSIO3), dma_setup(WSIO3)
NAME
bp_dma_setup(WSIO3) – Set up a DMA transfer for a list of buffers.

SYNOPSIS

int bp_dma_setup (struct isc_table_type *isc, struct buf *bp,
                 struct bp_dma_parms *bp_dma_parms);

PARAMETERS

isc Pointer to an ISC structure.
bp Pointer to the first buf structure.
bp_dma_parms Pointer to a dma_parms structure.

DESCRIPTION

The bp_dma_setup() WSIO sets up a DMA transfer for a list of buffers. A driver calls this function when bp->b_merge_cnt is not zero (i.e., when the B2_LIST flag is set in bp->b2_flags). Buffer lists are sent to the driver only when the driver registers that it can handle B2_LIST buffers. The bp_dma_parms structure must be initialized by the driver before calling bp_dma_setup().

RETURN VALUES

0 Successful completion.
<>0 Error.

CONSTRAINTS
EXAMPLES

```c
struct bp_dma_parms *bp_dma_parms = &lsp->bp_dma_parms;
struct dma_parms *dma_parms = &bp_dma_parms->dma_parms;

dma_parms->flags = NO_WAIT;
dma_parms->channel = BUS_MASTER_DMA;
dma_parms->dma_options = DMA_8BYTE;

if (bp->b_merge_cnt == 0) {
    dma_parms->dma_options = (bp->b_flags & B_READ) ?
        DMA_READ : DMA_WRITE;
    dma_parms->spaddr = bp->b_spaddr;
    dma_parms->addr = bp->b_un.b_addr;
    dma_parms->count = bp->b_bcount;
    retval = dma_setup(isc, dma_parms);
} else {
    retval = bp_dma_setup(isc, bp, bp_dma_parms);
}
```

SEE ALSO

hp_dma_cleanup(WSIO3), dma_cleanup(WSIO3), dma_setup(WSIO3)
NAME

dma_cleanup(WSIO3) – Clean up from a DMA transfer.

SYNOPSIS

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 for a
DMA transfer.

RETURN VALUES

None.

CONSTRAINTS

SEE ALSO

hp_dma_setup(WSIO3), 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;
    space_t spaddr;
    int count;
};

#include <sys/isa.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
#define DMA_16BYTE 0x400
#define DMA_32BYTE 0x800
#define DMA_READ 0x1000
#define DMA_WRITE 0x2000

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

bp_dma_cleanup(WSIO3), bp_dma_setup(WSIO3),
dma_cleanup(WSIO3), dma_setup(WSIO3), iovec(KER4),
wsio_map(WSIO3)
NAME

dma_setup(WSIO3) – Set up a DMA transfer

SYNOPSIS

#include <wsio/wsio.h>

int * dma_setup (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_setup() WSIO function sets up a DMA transfer. The
dma_Parms structure must be initialized before calling dma_setup().

RETURN VALUES

0 Successful completion.

<>0 Error.

CONSTRAINTS
EXAMPLES

struct dma_parms *thisdma;

/*
 * Common DMA setup code. Most of this code does not
 * vary between READ and WRITE. The dma channel has been
 * allocated during attach/init so that does not have to
 * be done here.
 */
thisdma = &pdp->pd_dma_parms;
bzero(thisdma, sizeof(struct dma_parms));
thisdma->flags = NO_WAIT;
thisdma->channel = BUS_MASTER_DMA;
thisdma->addr = bp->b_un.b_addr;
thisdma->spaddr = bp->b_spaddr;
thisdma->count = bp->b_bcount;
thisdma->dma_options = DMA_8BYTE;

if (bp->b_flags & B_READ) {
    thisdma->dma_options |= DMA_READ;
} else {
    thisdma->dma_options |= DMA_WRITE;
}

if (dma_setup(pdp->isc, thisdma) == 0) {
    /* DMA setup successfully completed. */
}

SEE ALSO

bp_dma_setup(WSIO3), dma_cleanup(WSIO3)
NAME

`driver_addr_probe(WSIO_DRV)` – Interface driver address probing function.

SYNOPSIS

```c
#include <wsio/wsio.h>
#include <sys/ioparams.h>

int 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);
```

PARAMETERS

- `this_node` A pointer to an `io_tree_node` struct.
- `dev_probe` Probe function registered by device driver to be called by `driver_addr_probe()`.
- `drv_info` The `drv_into_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.
- `name` 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

- 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

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

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.

See HP-UX Driver Development Guide for details;
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.

CONSTRAINTS

EXAMPLES

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 drivers:

```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);
    }

    SEE ALSO

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

driver_close(WSIO_DRV) – Close a device

SYNOPSIS

#include<sys/conf.h>

int driver_close (dev_t dev, int flag, int mode);

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).

mode Determines whether this is a call to a block or char
driver. This parameter is not accessible from a close (2)
call.

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.

**CONSTRAINTS**

**SEE ALSO**

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

driver_dev_init(WSIO_DRV) – Initialize a device driver

SYNOPSIS

int driver_dev_init(void);

PARAMETERS

None.

DESCRIPTION

The 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, dev_init(), with the driver_install() routine. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN VALUES

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

CONSTRAINTS

SEE ALSO

driver_install(WSIO_DRV)
NAME

driver_dev_probe(WSIO_DRV) – Interface driver dev probing function.

SYNOPSIS

#include <wsio/wsio.h>
#include <sys/ioparams.h>

int driver_dev_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 *name, 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.

name 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_dev_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_dev_probe() routine as part of your driver_install() routine. Commonly, driver is replaced by your driver's name.

See HP-UX Driver Development Guide for details;

RETURN 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 (WSIO3),
wsio_probe_dev_info (WSIO4), wsio_register_dev_func (WSIO3)
NAME

driver_if_init(WSIO_DRV) – Initialize interface driver

SYNOPSIS

#include<sys/io.h>

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(void);

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

driver_ioctl(WSIO_DRV) – Execute driver-specific control functions

SYNOPSIS

#include<sys/conf.h>

int driver_ioctl (dev_t dev, int cmd, caddr_t arg_ptr, int flag);

PARAMETERS

dev Device number.

cmd Command word.

arg_ptr Pointer to the command word arguments, if any.

flag File access flags.

DESCRIPTION

The 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 d_ioctl 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

0 Successful completion

<> 0 Error. The value is expected to be an error value.

CONSTRAINTS

LP64 CONSIDERATIONS

Pay particular attention to the 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

```c
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

- 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 <wsio/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

>0                    Successful completion. The value is the minor number.

-1                    Error. The minor number could not be built.

CONSTRAINTS
SEE ALSO

wsio_drv_data_t(WSIO4)
NAME

`driver_minphys(WSIO_DRV)` – Driver specific transfer size adjustment

SYNOPSIS

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

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

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.

```c
void mydriver_minphys(struct buf *bp) {
    if (bp->b_bcount > NBPG)
        bp->b_bcount = NBPG;
}
```

CONSTRAINTS
SEE ALSO

minphys(KER2), physio(KER2)
NAME

driver_open(WSIO_DRV) – Open a device

SYNOPSIS

#include<sys/conf.h>

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.

  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

#include<sys/conf.h>

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

>0 Successful completion. The value is the swap partition size.

-1 Error.

CONSTRAINTS

SEE ALSO

drv_ops (CD104)
NAME

driver_read(WSIO_DRV) – Read data from/to a character device

SYNOPSIS

#include<sys/conf.h>

int driver_read (dev_t dev, struct uio * uio);

PARAMETERS

dev The device number of the associated device file. The
routine can extract the major and minor numbers from
the device number. Your driver_open() routine
should verify that the minor number is valid.

uio A pointer to a uio structure. The uio structure
contains information about the data being read or
written.

DESCRIPTION

When a user process issues a read(), readv(), write(), or writev() system call for a character device, the kernel puts information about the request in the uio and iovec structures and dispatches control to the
driver_read() or driver_write() routine for that device, passing the
uio structure to the driver as a parameter.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion.

<>0 Error. The value is expected to be an errno value.

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

#include<sys/conf.h>

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

<>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 `u.u_error`. If it detects an error while selecting for an exception condition, it should return true and set an error in `u.u_error`.

**CONSTRAINTS**

**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

#include<sys/conf.h>

void driver_strategy (struct buf * bp);

PARAMETERS

bp Pointer to a buf structure.

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

driver_write(WSIO_DRV) – Write data from/to a character device

SYNOPSIS

#include<sys/conf.h>

int driver_write (dev_t dev, struct uio * uio);

PARAMETERS

dev The device number of the associated device file. The routine can extract the major and minor numbers from the device number. Your driver_open() routine should verify that the minor number is valid.

uio A pointer to a uio structure. The uio structure contains information about the data being read or written.

DESCRIPTION

When a user process issues a read(), readv(), write(), or writev() system call for a character device, the kernel puts information about the request in the uio and iovec structures and dispatches control to the driver_read() or driver_write() routine for that device, passing the uio structure to the driver as a parameter.

See HP-UX Driver Development Guide for details;

RETURN VALUES

0 Successful completion.

<>0 Error. The value is expected to be an errno value.

CONSTRAINTS
SEE ALSO

drv_ops(CDIO4), physio(KER2), uiomove(KER2)
NAME

free_isc(WSIO3) – Free a driver's ISC entry

SYNOPSIS

#include <wsio/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

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><code>bus_type</code></td>
<td>Copied from <code>old_isc</code></td>
</tr>
<tr>
<td><code>my_isc</code></td>
<td>Copied from <code>old_isc</code></td>
</tr>
</tbody>
</table>
get_new_isc(WSIO3)

- `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**

- `<>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_iiodone. 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 >= BUF_HASH_LOCK_ORDER.
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

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

```c
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 <wsio/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

```c
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
isc_table_type(WSIO4) – ISC table entry structure

SYNOPSIS
#include <sys/io.h>

DESCRIPTION
Each interface card (each device claimed by an interface driver) has an associated Interface Select Code (ISC) entry. The contents of an ISC entry are declared as the isc_table_type structure in <sys/io.h>.

WSIO uses the pointer to an ISC entry as the handle to an interface card. Many WSIO functions require the pointer to an ISC as an argument. WSIO also passes card-specific information to an interface driver through fields in the ISC entry.

Interface drivers use the ISC to store driver-specific information. Some fields in the ISC entry are defined by system and drivers are expected to use these fields as intended by the system. Other fields are available to the driver for its internal use.

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 to specify communication between device and interface drivers.

Generic Function Switch
The generic function switch, isc->gfsw, is intended for system-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. Its usage is currently not implemented, and drivers must set this pointer to NULL.
Interface Function Switch

The interface function switch, `isc->ifsw`, is intended for device-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 `scsi_ifsw` structure in `<wsio/scsi_ctl.h>`.

STRUCTURE MEMBERS

The following is a list of driver accessible fields in the ISC structure that are initialized by the system.

Table 4-1: Driver Relevant `isc_table_type` Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>bus_type</code></td>
</tr>
<tr>
<td>int</td>
<td><code>if_id</code></td>
</tr>
<tr>
<td>caddr_t</td>
<td><code>if_info</code></td>
</tr>
<tr>
<td>int</td>
<td><code>if_info-&gt;flags</code></td>
</tr>
<tr>
<td>caddr_t</td>
<td><code>if_reg_ptr</code></td>
</tr>
</tbody>
</table>

- **bus_type**: Type of I/O bus for the interface card. For example, `PCI_BUS` for PCI interface cards.
- **if_id**: Hardware ID of the interface card. The contents of this field are `bus_type` dependent.
- **if_info**: Pointer to a `wsio_if_info` structure declared in `<sys/wsio.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.

The following is a list of fields in the ISC structure that are initialized by drivers.

**Table 4-2  Driver Initialized ISC Fields**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile int *</td>
<td>card_ptr</td>
</tr>
<tr>
<td>int (*)(struct isc_table_type *)</td>
<td>gfsw-&gt;init</td>
</tr>
<tr>
<td>caddr_t</td>
<td>if_drv_data</td>
</tr>
<tr>
<td>void *</td>
<td>if_isc</td>
</tr>
<tr>
<td>caddr_t</td>
<td>ifsw</td>
</tr>
</tbody>
</table>
Table 4-2  Driver Initialized ISC Fields (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>my_address</td>
</tr>
<tr>
<td>struct buf *</td>
<td>owner</td>
</tr>
<tr>
<td>unsigned int</td>
<td>state</td>
</tr>
</tbody>
</table>

**card_ptr**  Pointer to a range of memory mapped interface card registers.

**gfsw->init**  Pointer to the `init` function for the interface driver. The system initializes `gfsw` to point to a generic function switch table. The interface driver is responsible for updating the table.

**if_drv_data**  Pointer to a driver specified object.

**if_isc**  Pointer to a driver specified object.

**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.

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

**owner**  Can be used as desired. Usually contains a pointer to the active `buf` or I/O request.

**state**  Can be used as desired. Usually contains the device state information.
NAME

m_wsio_funcnum(WSIO3) – Get the number of an interface card function

SYNOPSIS

#include<sys/wsio.h>

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
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
m_wsio_selcode(WSIO)

NAME

m_wsio_selcode(WSIO3) – Get the select code for a device

SYNOPSIS

#include<sys/wsio.h>

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
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

\texttt{m\_wsio\_vsc} (WSIO3) – Return the system bus module number for a device number

SYNOPSIS

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

int m\_wsio\_vsc (dev\_t dev, wsio\_drv\_info\_t * drv\_hdr\_ptr);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \texttt{dev} \hspace{1cm} The dev\_t number of a device.
  \item \texttt{drv\_hdr\_ptr} \hspace{1cm} Pointer to the wsio\_drv\_info\_t structure for the device.
\end{itemize}

DESCRIPTION

The \texttt{m\_wsio\_vsc} WSIO function returns the system bus module number for device number \texttt{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_wsioAttachListAdd() WSIO function adds the driver attach function pointer attach_func to the WSIO attach list specified by type.

Dynamically loadable drivers call mod_wsioAttachListAdd() 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

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

`mod_wsio_attach_list_remove(WSIO)` – Remove the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

```c
int mod_wsio_attach_list_remove (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_remove()` WSIO function removes the driver attach function pointer `attach_func` from the WSIO attach list specified by `type`. The `attach_func` and `type` parameters must match the parameters passed to `mod_wsio_attach_list_add()`.

Dynamically loadable drivers call `mod_wsio_attach_list_remove()` when they are unloaded from 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.

RETURN VALUES

- 0 Successful completion.
- 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

```c
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

```c
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_alloc_mem`(WSIO3) – Service to allocate memory for DMA buffers or control structures.

SYNOPSIS

```c
wsio_alloc_status_t
wsio_alloc_mem ( wsio_mem_handle_t mem_handle,
    size_t size,
    wsio_vaddr_t * vaddr,
    wsio_alloc_flags_t flags)
```

PARAMETERS

- **mem_handle**: A handle allocated by a call to `wsio_alloc_mem_handle`
- **size**: The size of the buffer.
- **vaddr**: A pointer to the allocated buffer if successful else NULL
- **flags**: Flags which describe restrictions
  - `WSIO_SLEEP_OK`: Flag to indicate service can sleep if waiting for resources

DESCRIPTION

This WSIO service can be called by drivers to allocate memory for DMA buffers or control structures. The first parameter to the service must be a `mem_handle` that was allocated by the driver using `wsio_alloc_mem_handle`.

RETURN VALUES

- `WSIO_ALLOC_OK`  
  Indicates that the buffer was allocated
- `WSIO_ALLOC_OUT_OF_RESOURCES`  
  Unable to allocate the specified resources
NAME

wsio_alloc_mem_handle(WSIO3) – Service to specify the type of memory to allocate.

SYNOPSIS

wsio_alloc_status_t
wsio_alloc_mem_handle ( struct isc_table_type *isc,
                        wsio_mem_handle_t* mem_handle,
                        wsio_mem_alloc_attrib_t attribs)

PARAMETERS

isc A pointer to the device's isc structure
mem_handle A pointer to a variable where the handle will be stored upon completion.
attrs Attributes describing the criteria for the type of memory to allocate.

WSIO_OPTIMIZE_FOR_DEVICE
Allocate memory close to the device. On Half-Dome systems, memory will be allocated on the same cell as the device.

WSIO_OPTIMIZE_FOR_CPU
Allocate memory close to the current CPU. On Half-Dome systems, memory will be allocated on the same cell as the CPU. This is the default behavior.

WSIO_32BIT_MEMORY
The buffer must be allocated below 4G.

WSIO_IO_CONTIGUOUS
On platforms without an IOpdir physically contiguous memory will be allocated.
WSIO_ALIGN_ON_SIZE

With this attribute, buffers will be aligned with the same alignment as their size.

DESCRIPTION

Drivers will call this service to specify the type of memory they want to allocate. The service will return a *mem_handle* which can be passed into the WSIO memory alloc and free routines. Drivers can allocate more than one *mem_handle* to specify different criteria for memory allocation.

While it is not a requirement, it is recommended that drivers call this routine early in their initialization sequence. This is due to the high overhead of the routine. Buffer alignment is as follows:

<table>
<thead>
<tr>
<th>Allocation Size</th>
<th>Buffer Aligned On</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Than Cacheline Size</td>
<td>32 Byte Boundary</td>
</tr>
<tr>
<td>Greater Than or Equal to Cacheline Size</td>
<td>Cacheline Boundary</td>
</tr>
<tr>
<td>Greater Than or Equal to I/O Page Size (4K)</td>
<td>4K Boundary</td>
</tr>
</tbody>
</table>

RETURN VALUES

**WSIO_ALLOC_OK**
Indicates that the buffer was allocated

**WSIO_ALLOC_OUT_OF_RESOURCES**
Unable to allocate the specified resources

**WSIO_INVAL_PARAM**
A parameter was not valid

CONSTRAINTS

None

EXAMPLES
SEE ALSO
NAME

wsio_allocate_dma_handle(WSIO3) – Obtain a handle used to setup DMA

SYNOPSIS

#include <wsio/wsio.h>

void *wsio_allocate_dma_handle (struct isc_table_type *isc);

PARAMETERS

isc Pointer to the driver’s isc_table entry.

DESCRIPTION

The wsio_allocate_dma_handle() WSIO function is called by device drivers to obtain a DMA handle. This handle, which is passed to all DMA services, can be associated with various DMA hints, and is used to control DMA. Multiple handles can be allocated, allowing a device driver to associate different hints with each handle.

RETURN VALUES

A void pointer to the handle. If NULL is returned, a handle count could not be allocated.

CONSTRAINTS

EXAMPLE

void *dma_handle;

DMA_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL { 
    /* No handle allocated. */
    return (ERROR);
}
/* The DMA handle is now in the dma_handle variable */
SEE ALSO

wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3)
NAME

\texttt{wsio\_allocate\_shared\_mem(WSIO3)} – Set up an I/O virtually contiguous DMA buffer.

SYNOPSIS

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

wsio_map_status_t wsio_allocate_shared_mem(
    struct isc_table_type *isc, void *dma_handle,
    size_t size, wsio_iova_t *iova, wsio_vaddr_t *vaddr,
    wsio_shared_mem_attr_t shared_mem_attr);
\end{verbatim}

PARAMETERS

- \texttt{isc} \hspace{1cm} Pointer to the driver's \texttt{isc\_table} entry.
- \texttt{dma\_handle} \hspace{1cm} DMA handle allocated using \texttt{wsio\_allocate\_dma\_handle()}. 
- \texttt{size} \hspace{1cm} Size of buffer to allocate.
- \texttt{iova} \hspace{1cm} Pointer that contains the I/O virtual address upon completion. A \texttt{wsio\_iova\_t} must be allocated by the driver, and the pointer to this is what should be passed into the macro.
- \texttt{vaddr} \hspace{1cm} Pointer that contains the host virtual address upon completion. A \texttt{wsio\_vaddr\_t} must be allocated by the driver, and the pointer to this is what should be passed into the macro.
- \texttt{shared\_mem\_attr} \hspace{1cm} Bitmask that indicates how to allocate the memory. The acceptable values are described in the following list. If a type of 0 is used, the default behavior of \texttt{WSIO\_IO\_SHMEM\_OPTIMIZE\_DEVICE\_LATENCY} is used. The behavior of the allocation is also affected by attributes set using \texttt{wsio\_dma\_set\_device\_attributes()}, and \texttt{wsio\_set\_dma\_attributes()}. 


The following are the `wsio_shared_mem_attr_t` allowable bitmask values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WSIO_IO_SHMEM_OPTIMIZE_DEVICE_LATENCY</code></td>
<td>Allocation should optimize for device access latency. If possible, allocate object in memory local to a bus bridge.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_OPTIMIZE_HOST_LATENCY</code></td>
<td>Allocation should optimize for host access latency. If possible, allocate in host memory.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_DMA_ALLOC_COMPATIBLE</code></td>
<td>Behave exactly as the 10.X <code>dma_alloc</code> service. This type is for compatibility with 10.X only.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_INBOUND</code></td>
<td>This attribute can be OR’ed with the other attributes to indicate the buffer is used exclusively for inbound DMA.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_OUTBOUND</code></td>
<td>This attribute can be OR’ed with the other attributes to indicate the buffer is used exclusively for outbound DMA.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_DEV_WEAK_OK</code></td>
<td>This attribute can be OR’ed with the other attributes to indicate the accesses to the buffer can be weakly ordered. The default ordering is the strongest that can be provided for the given I/O bus.</td>
</tr>
<tr>
<td><code>WSIO_IO_SHMEM_ALIGN_ON_SIZE</code></td>
<td>This attribute can be OR’ed with the other attributes to specify <code>size</code> also indicates the alignment boundary for the allocation.</td>
</tr>
</tbody>
</table>
DESCRIPTION

The \texttt{wsio\_allocate\_shared\_mem()} WSIO function is called by a device driver to allocate an I/O virtually contiguous DMA buffer that is to be used for continuous DMA. Continuous DMA means that the memory appears contiguous to the I/O device and can be read or written by the I/O device on a continuous basis. For packet DMA or DMA that is used for temporary mappings, \texttt{wsio\_map\_dma\_buffer()} should be used.

If a callback function is set up (see \texttt{wsio\_set\_dma\_callback()}) and no resources are available when the call is made, \texttt{WSIO\_MAP\_W\_CALLBACK} will be returned, and the callback will be triggered when resources become available.

This macro may be called in a non-blocking context.

RETURN VALUES

\begin{itemize}
\item \textbf{WSIO\_MAP\_OK} \hspace{1em} Success.
\item \textbf{WSIO\_MAP\_W\_CALLBACK} \hspace{1em} Returned if no resources are available and a callback is registered.
\item \textbf{WSIO\_MAP\_E\_NO\_RESOURCES} \hspace{1em} Returned if no resources are available and no callback is registered.
\item \textbf{WSIO\_MAP\_E\_RESOURCE\_ERROR} \hspace{1em} Returned if cannot allocate resources. If this is returned, the allocation will never succeed.
\item \textbf{WSIO\_MAP\_E\_PARAMETER\_ERROR} \hspace{1em} Returned on bad parameter (Software bug).
\item \textbf{WSIO\_MAP\_E\_UNKNOWN\_ERROR} \hspace{1em} Returned if there is an unknown error.
\end{itemize}

CONSTRAINTS
EXAMPLE

```c
void *dma_handle = NULL;
wsio_iova_t io_virtual_addr;
wsio_vaddr_t host_virtual_addr;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return ERROR;
}

/* The DMA handle is now in the dma_handle variable */
if (wsio_allocate_shared_mem(isc_entry,dma_handle,buf_size,
    &io_virtual_addr,&host_virtual_addr,0) != WSIO_MAP_OK) {
    /* Unable to allocate shared memory, so return an error */
    return ERROR;
}

/* A buffer of size 'buf_size' is now allocated and mapped
   * in both host virtual memory and I/O virtual memory space.
   */
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_flush_shared_mem(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3)
NAME

`WSIO_BIG_ENDIAN` (WSIO3) – Function to return true (1) if the local bus is big-endian.

SYNOPSIS

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

int WSIO_BIG_ENDIAN (struct isc_table_type *isc);
```

PARAMETERS

- `isc` Pointer to the driver's `isc_table` entry.

DESCRIPTION

The `WSIO_BIG_ENDIAN()` macro is called by a device driver to report whether the local bus is big-endian. If it is, true is returned, otherwise it returns false. This can be used by a driver along with the known endianness of the host processor to decide whether endian swapping should be performed. Endian swapping might be necessary for any data transfers between the I/O bus and local host memory.

`WSIO_BIG_ENDIAN()` can be called in a non-blocking context.

RETURN VALUES

- 1 Local bus is big-endian.
- 0 Not big-endian

CONSTRAINTS

EXAMPLE

```c
if (WSIO_BIG_ENDIAN(isc_entry)) {
    /* No endian swapping necessary */
} else {
    /* Endian swapping must be performed */
}
```
SEE ALSO

WSIO_LITTLE_ENDIAN(WSIO3)
NAME

wsio_cfg_inXX(WSIO3) – Macros to read from configuration space.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_cfg_inXX (struct isc_table_type *isc,
                    wsio_addr_handle_t cfg_handle,
                    uint32_t offset, uintXX_t *data);

PARAMETERS

isc Pointer to the driver's isc_table entry.

 cfg_handle Configuration space handle.

 offset Byte offset into the configuration space.

 data Pointer to an appropriately sized and aligned memory space for the returned data.

DESCRIPTION

The wsio_cfg_inXX() macros are called by device drivers to read from configuration space. The cfg_handle and the offset are used to specify the correct location to read from. The value ‘XX’ refers to 8, 16, 32, or 64 and indicates the amount of data to read from configuration space. Endian translation is performed automatically if the host memory and local bus have different endianness.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t handle;
uint32_t data;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    return(ERROR);
}
wsio_cfg_in32(isc_entry,handle,offset,&data);

/* 'data' will now contain whatever was at 'offset' in
 * configuration space
 */
```

SEE ALSO

wsio_cfg_outXX(WSIO3), wsio_map_cfg_handle(WSIO3),
wsio_unmap_cfg(WSIO3)
NAME

\texttt{wsio\_cfg\_outXX(WSIO3)} – Macros for writing to configuration space

SYNOPSIS

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

void wsio_cfg_outXX (struct isc_table_type *isc,
                     wsio_addr_handle_t cfg_handle,
                     uint32_t offset, uintXX_t data)
\end{verbatim}

PARAMETERS

- \texttt{isc} Pointer to the driver's \texttt{isc\_table\_entry}.
- \texttt{cfg\_handle} Configuration handle.
- \texttt{offset} Byte offset into the configuration space.
- \texttt{data} Pointer to an appropriately sized and aligned memory space for the returned data.

DESCRIPTION

The \texttt{wsio\_cfg\_outXX()} macros are called by device drivers to write to configuration space. The \texttt{cfg\_handle} and the \texttt{offset} are used to specify the correct location to write to.

The value 'XX' refers to 8, 16, 32, or 64 and indicates the amount of data to write to configuration space. Endian translation is performed automatically if host memory and the local bus have different endianness.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

wsio_addr_handle_t handle;
uint32_t data = 0x5a;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    return(ERROR);
}
wsio_cfg_out32(isc_entry,handle,offset,data);

/* 0x5a will now be at 'offset' in configuration space */

SEE ALSO

wsio_cfg_inXX (WSIO3), wsio_map_cfg_handle (WSIO3),
wsio_unmap_cfg (WSIO3),
NAME
wsio_dma_pass_thru(WSIO3) – Call a DMA pass-thru function that might not otherwise be accessible.

SYNOPSIS
#include <wsio/wsio.h>

intptr_t wsio_dma_pass_thru (struct isc_table_type *isc,
  void *dma_handle,
  wsio_pt_type_t pass_thru_type,
  intptr_t pass_thru_param);

PARAMETERS

isc Pointer to the driver’s isc_table entry.
dma_handle DMA handle allocated using
  wsio_allocate_dma_handle().

pass_thru_type Indicates which pass-thru function to call. Two types are defined as shown in the following items:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MAP_PT_DEVICE_LOCK</td>
<td>Prevents a device’s access to shared memory. This can be used so that processors can access host memory atomically, and can be used for synchronization. The parameter’s pass_thru_param indicates whether shared memory should be locked (1) or unlocked (0). Implementation of this function is not required, so the return code is zero (0) if the function is implemented or non-zero if not implemented.</td>
</tr>
</tbody>
</table>


**DESCRIPTION**

The `wsio_dma_pass_thru()` WSIO function is present to allow new DMA interfaces to be added without breaking binary compatibility. It allows such interfaces to be accessed if they are present.

**RETURN VALUES**

Dependent on the specific pass-thru function being called.

**CONSTRAINTS**

**EXAMPLE**

```c
if (wsio_dma_pass_thru(isc_entry,dma_handle,
    WSIO_MAP_PT_SYNC_BUSSES,0) !=0) {
    /* The function isn’t implemented */
    return(ERROR);
} else {  
    /* The function is implemented, and completed correctly */
    return(0);
}
/* This code will attempt to sync memory associated with 
* dma_handle 
*/
```
SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3)
NAME

wsio_dma_set_device_attributes(WSIO3) – Associate DMA hints with a device

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_dma_set_device_attributes
    (struct isc_table_type * isc,
     void * dma_handle,
     wsio_dma_attribute_t attribute
     wsio_dma_attr_param_t param);

PARAMETERS

isc Pointer to the driver’s isc_table entry.
dma_handle DMA handle allocated using
    wsio_allocate_dma_handle().
attribute Indicates which hint to set for the device associated
    with dma_handle. The possible attributes follow:

WSIO_DMA_ATTR_ADDR_WIDTH
    Bits of addressing supported by the
    device. This is used to determine
    whether a device can DMA directly to
    memory buffers.
    Default value = 32

WSIO_DMA_ATTR_ALIGNMENT
    Byte alignment of DMA buffer
    required for device.
    Default value = HW Dep.
WSIO DMA ATTR ATM
ATM hint. Used by hardware in some implementations.

0 = not ATM
1 = ATM48 (optimize for 48-byte transfers)
2 = ATM192 (optimize for 192-byte transfers)

Default value = 0

WSIO DMA ATTR CALLBACK
Specifies a function to call when resources become available.

Default value = NULL

WSIO DMA ATTR CALLBACK ARG
Specifies an argument to the callback function.

Default value = 0

WSIO DMA ATTR_FLUSH_ON_USE
Specifies the cacheline should be flushed from any intermediate buffers as soon as it is referenced. This inhibits any coalescing of data by bus bridges.

Default value = 0

WSIO DMA ATTR_IGN_ALIGN
Specifies the mapping service should not handle cacheline fragments in a special way.

Default value = 0
WSIO_DMA_ATTR_INTERLEAVE
IOVA allocation model

0 = DMA streams are normally interleaved (mass-storage).
1 = DMA streams are normally not interleaved (networking)
2 = DMA buffers are static and accessed randomly (low fat).

Default value = 0

WSIO_DMA_ATTR_PREFETCH
Specifies how aggressively hardware should prefetch for outbound DMA.

0 = no prefetch
1 = moderate prefetch
2 = aggressive prefetch

Default value = 1

WSIO_DMA_ATTR_SAFE
Specifies the most conservative coherency model should be used for inbound DMA. Inhibits semicoherent transactions such as WRITE_PURGE unless it is guaranteed that no data in processor caches will be lost.

1 = ON
2 = OFF

Default value = 0

WSIO_DMA_ATTR_TXN_SIZE
Specifies the default transaction size used by the device. This is used by hardware to optimize conversion of transactions between buses.

Default value = HW Dep.

WSIO_DMA_ATTR_INBOUND
DMA buffers will be used exclusively for inbound DMA.

Default value = 0
DESCRIPTION

The `wsio_dma_set_device_attributes()` WSIO function is used to associate DMA transaction hints and attributes with a specific device. These hints are overridden by any hints set for a specific DMA handle via `wsio_set_dma_attributes()`, or some hints passed in as parameters to `wsio_map_dma_buffer()`.

`wsio_dma_set_device_attributes()` can be called in a non-blocking context. If `WSIO_DMA_ATTR_INTERLEAVE` is set to 1, a subsequent `wsio_allocate_shared_mem()` or `wsio_map_dma_buffer()` can only successfully request a buffer with a maximum of one page (4K) in length and this buffer cannot cross a page boundary. This is the limitation placed by the underlying platform. If a larger buffer is desirable, use the default value of 0. This larger buffer can be used for control structures rather than packet DMAs.

Do not call `wsio_dma_set_device_attributes()` to set the `WSIO_DMA_ATTR_INTERLEAVE` to the default value of 0. The call will fail.

RETURN VALUES

- `WSIO_MAP_OK` Success.
- `WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call.

`param` Information dependent on the hint or attribute being set. Check the attribute list for more information.
CONSTRAINTS

EXAMPLE

```c
if (wsio_dma_set_device_attributes(isc_entry, dma_handle, 
    WSIO_DMA_ATTR_INTERLEAVE, 1) != WSIO_MAP_OK) {
    /* There was a parameter error */
    return(ERROR);
} else {
    /* DMA streams are now not normally interleaved for all DMA 
      * associated with this device 
      */
    return(0);
}
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3),
NAME

`wsio_drv_data_t` (WSIO4) – Driver-specific fields for WSIO drivers

SYNOPSIS

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

DESCRIPTION

The `wsio_drv_data_t` WSIO structure type, defined in `<wsio/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><code>drv_path</code></td>
<td>Follow these guidelines:</td>
</tr>
<tr>
<td></td>
<td>- For device drivers, <code>drv_path</code> is typically a string that contain the interface card's type and the device's class. For example, <code>scsi_disk</code>.</td>
</tr>
<tr>
<td></td>
<td>- For interface drivers, <code>drv_path</code> should match the card's type. For example, <code>scsi</code>.</td>
</tr>
<tr>
<td></td>
<td>- For pseudo drivers, <code>drv_path</code> should match the card's class. For example, <code>graphics</code>.</td>
</tr>
<tr>
<td><code>drv_type</code></td>
<td>One of the following values:</td>
</tr>
<tr>
<td><code>T_INTERFACE</code></td>
<td>The driver controls an interface card.</td>
</tr>
<tr>
<td><code>T_DEVICE</code></td>
<td>The driver controls a hardware device.</td>
</tr>
<tr>
<td><code>drv_flags</code></td>
<td>One of the following values:</td>
</tr>
<tr>
<td><code>DRV_CONVERGED</code></td>
<td>The driver meets the HP-UX Release 10.0 Converged I/O specifications. All new drivers should meet these specifications.</td>
</tr>
<tr>
<td><code>NOT_CONVERGED</code></td>
<td>The driver conforms to the pre-Release 10.0 unconverged specifications.</td>
</tr>
</tbody>
</table>
**wsio_drv_data_t**

**wsio_drv_data_t**(WSIO4)

**CHANNEL 4 275**

**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,
};
```

### SEE ALSO
NAME

\texttt{wsio_drv_info} (WSIO4) – Structure containing pointers to other CDIO and WSIO data structures

SYNOPSIS

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

DESCRIPTION

The \texttt{wsio_drv_info_t} WSIO structure type, defined in \texttt{<wsio/wsio.h>}, contains pointers to three other data structures.

STRUCTURE MEMBERS

\begin{itemize}
\item \texttt{drv_info} Pointer to a \texttt{drv_info_t} CDIO structure.
\item \texttt{drv_ops} Pointer to a \texttt{drv_ops_t} CDIO structure.
\item \texttt{drv_data} Pointer to a \texttt{wsio_drv_data_t} structure.
\item \texttt{driver_version} Set to \texttt{WSIO_DRV_CURRENT_VERSION}.
\end{itemize}

SEE ALSO

\begin{verbatim}
drv_info(CDIO4), drv_ops(CDIO4), wsio_drv_data_t (WSIO4)
\end{verbatim}
NAME

wsio_event_t(WSIO5) – An enumeration of WSIO associated events

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This structure enumerates the WSIO associated events. When the driver handler is invoked for a WSIO event, the argument passed to the driver handler is wsio_generic_event_t type. The event field in the argument structure indicates the event for which the driver handler is invoked, as described in the Example section.

STRUCTURE MEMBERS

typedef enum {
    WSIO_NO_EVENT,
    WSIO_EVENT_SUSPEND,
    WSIO_EVENTResume,
    WSIO_EVENT_REMOVE,
    WSIO_EVENT_DEV_ERROR,
    WSIO_EVENT_BUS_ERROR,
    WSIO_EVENT_SELF_TEST,
    WSIO_EVENT_LBI_INTR_MIGR,
    WSIO_EVENT_OFFLINE_CPU,
    WSIO_EVENT_ONLINE_CPU
} wsio_event_t;

EXAMPLE

If the driver has a transaction based interrupt, the driver is notified if that interrupt is being reassigned to a new CPU. The wsio_generic_event_t structure is filled with the following information:

<table>
<thead>
<tr>
<th>event</th>
<th>WSIO_EVENT_OFFLINE_CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_id</td>
<td>WSIO provided event_id</td>
</tr>
</tbody>
</table>
isc         Pointer to the instance of the associated driver isc_table_type structure

wsio_completion_cb         WSIO provided call back
arg         Pointer to a structure of type wsio_intr_migr_t

Refer to the Interrupt Migration chapter of the Driver Development Guide for relevant information.

SEE ALSO

wsio_drv_event_t(WSIO5),
wsio_install_drv_event_handler(WSIO3),
wsio_reg_drv_capability_mask(WSIO5)
NAME

wsio_fastmap_dma_buffer(WSIO3) – Function to map an existing memory object for packet DMA.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_fastmap_dma_buffer (  
    struct isc_table_type * isc,  
    void * dma_handle,  
    wsio_range_type_t range_type,  
    wsio_dma_map_t * host_range,  
    wsio_dma_map_t * io_range);

PARAMETERS

isc Pointer to the driver’s isc_table entry.
dma_handle DMA handle allocated using  
    wsio_allocate_dma_handle().
range_type Indicates the type of host memory being mapped. It can be:
    KERNELSPACE Indicates host_range is a kernel virtual buffer.
    PHYSICAL Indicates host_range is a physical buffer.
    > 0 Indicates host_range is in user space, and this will be the space ID of the virtual address.
host_range Pointer to an address/length structure that contains information about the host space to map. If the mapping was only partially completed, this will contain information about the remaining space to be mapped when the call completes.
io_range Pointer to an address/length structure that will contain information about the I/O space that was mapped.
DESCRIPTION

The `wsio_fastmap_dma_buffer()` macro is called by a device driver to map an existing memory object for packet DMA. It operates in the same way as `wsio_map_dma_buffer()` except the entire host address range must reside on a single physical page. If the range cannot be mapped in a single call, an error will be returned. Cacheline fragments are ignored (the same behavior as `WSIO_DMA_IGN_ALIGNMENT`).

The size of a buffer can have a maximum length of one page (4K) and this buffer can not cross a page boundary.

All mappings will remain in effect until `wsio_unmap_dma_buffer()` or `wsio_remap_dma_buffer()` are called to remove or change them.

Device drivers can set up a callback routine that will come into play if resources are not available at the time a mapping is attempted. If this callback is set up, `WSIO_MAP_W_CALLBACK` will be returned instead of a no resource error. When resources become available, the callback routine will be called to indicate this to the device driver. For more information on how to setup and use a callback, see the `wsio_set_dma_callback()` manpage. `wsio_fastmap_dma_buffer()` can be called in a non-blocking context.

RETURN VALUES

`WSIO_MAP_OK` Returned if the entire buffer has been mapped.

`WSIO_MAP_W_CALLBACK` Returned if no resources are available and a callback function exists.

`WSIO_MAP_E_NO_RESOURCES` Returned if no resources are available and no callback function exists.

`WSIO_MAP_E_RESOURCE_ERROR` Returned if the request cannot and will never succeed.

`WSIO_MAP_E_HIGH_ADDR` Returned if the call failed because the device cannot reach the destination address.

`WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call.
WSIO MAP E_UNKNOWN_ERROR
Returned for hardware or other errors.

CONSTRAINTS

EXAMPLE

```c
void *dma_handle;
wsio_dma_map_t host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_fastmap_dma_buffer(isc_entry,dma_handle,
    KERNELSPACE,&host_range, &io_range)!= WSIO_MAP_OK) {
    /* Unable to map the range, so return an error */
    return(ERROR);
}

/* The host virtual buffer represented in the above code by
 * 'host_virtual_address' is now mapped. */
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_flush_shared_mem(WSIO3), wsio_init_map_context(WSIO3),
wsio_iova_to_phys(WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3),

Chapter 4 281
NAME

wsio_flush_shared_mem(WSIO3) – Flush an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <wsio/wsio.h>

wsio_map_status_t wsio_flush_shared_mem (
    struct isc_table_type *isc, void *dma_handle,
    size_t size, wsio_iova_t iova, wsio_vaddr_t vaddr,
    wsio_shared_mem_attr_t shared_mem_attr);

PARAMETERS

isc Pointer to the driver's isc_table entry.

dma_handle DMA handle allocated using
    wsioAllocate_dma_handle.

size Size of buffer to be flushed.

iova I/O virtual address of the shared memory.

vaddr Host virtual address of the shared memory.

shared_mem_attr Bitmask that was used to allocate the shared memory.

DESCRIPTION

The wsio_flush_shared_mem() WSIO function is called by a device
driver to guarantee the consistency of the memory object allocated via
wsioAllocate_shared_mem(). Any non-coherent buffers associated
with the memory object are flushed. All parameters to the function
should be the same as those passed to the call that allocated the memory.
wsio_flush_shared_mem() can be called in a non-blocking context.

RETURN VALUES

WSIO_MAP_OK Success.

WSIO_W_NOP The call has no effect. The caller need not call it again.
CON立场

EXAMPLE

```c
void *dma_handle = NULL;
wsio_iova_t io_virtual_addr;
wsio_vaddr_t host_virtual_addr;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return(ERROR);
}
/* The DMA handle is now in the dma_handle variable */

if (wsio_allocate_shared_mem(
    isc_entry, dma_handle, buf_size, &io_virtual_addr,
    &host_virtual_addr, 0) != WSIO_MAP_OK) {
    /* Unable to allocate the shared memory,
    * so return an error
    */
    return(ERROR);
}
/* A buffer of size 'buf_size' is now allocated and mapped in
* both host virtual memory and I/O virtual memory space.
*/
wsio_flush_shared_mem(isc_entry, dma_handle, buf_size,
                  io_virtual_addr, host_virtual_addr, 0);
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_free_dma_handle(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_iova_to_phys(WSIO3), wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3)
NAME

wsio_free_dma_handle(WSIO3) – Release a DMA handle.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_free_dma_handle (struct isc_table_type *isc, 
                          void *dma_handle);

PARAMETERS

isc Pointer to the driver's isc_table entry.

dma_handle Pointer to the DMA handle to free.

DESCRIPTION

The wsio_free_dma_handle() WSIO function is called by device drivers
to release a handle that has been allocated by
wsio_allocate_dma_handle(). It should be called anytime a handle is
no longer needed.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

```c
void *dma_handle;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return(ERROR);
}
/* The DMA handle is now in the dma_handle variable */

wsio_free_dma_handle(isc_entry, dma_handle);
/* The DMA handle has now been released */
```

SEE ALSO

`wsio_allocate_dma_handle(WSIO3)`, `wsio_allocate_shared_mem(WSIO3)`, `wsio_dma_pass_thru(WSIO3)`, `wsio_fastmap_dma_buffer(WSIO3)`, `wsio_free_shared_mem(WSIO3)`, `wsio_flush_shared_mem(WSIO3)`, `wsio_init_map_context(WSIO3)`, `wsio_iova_to_phys(WSIO3)`, `wsio_map_dma_buffer(WSIO3)`, `wsio_remap_dma_buffer(WSIO3)`, `wsio_set_device_attributes(WSIO3)`, `wsio_set_dma_attributes(WSIO3)`, `wsio_unmap_dma_buffer(WSIO3)`
NAME

\texttt{wsio\_free\_mem}(WSIO3) – Frees memory allocated by \texttt{wsio\_alloc\_mem}.

SYNOPSIS

\begin{verbatim}
void wsio_free_mem (wsio_mem_handle_t mem_handle,
                   wsio_vaddr_t vaddr)
\end{verbatim}

PARAMETERS

- \texttt{mem\_handle} A handle allocated by a call to \texttt{wsio\_alloc\_mem\_handle}.
- \texttt{vaddr} A pointer to the allocated buffer.

DESCRIPTION

This WSIO service is called to free memory allocated by the service \texttt{wsio\_alloc\_mem}.

RETURN VALUES

None.

CONSTRAINTS

EXAMPLE

SEE ALSO
NAME

wsio_free_mem_handle(WSIO3) – Destroy handle previously allocated by
wsio_alloc_mem_handle.

SYNOPSIS

void wsio_free_mem_handle (wsio_mem_handle_t mem_handle)

PARAMETERS

mem_handle A handle allocated by a call to
wsio_alloc_mem_handle.

DESCRIPTION

Drivers call this service to destroy a mem_handle that was allocated by a
previous call to wsio_alloc_mem_handle().

RETURN VALUES

None.

CONSTRAINTS

EXAMPLE

SEE ALSO
NAME

wsio_free_shared_mem(WSIO3) – Release an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_free_shared_mem (struct isc_table_type *isc,
                          void *dma_handle, size_t size,
                          wsio_iova_t iova, wsio_vaddr_t vaddr,
                          wsio_shared_mem_attr_t shared_mem_attr);

PARAMETERS

isc Pointer to the driver's isc_table entry.
dma_handle DMA handle allocated using wsio_allocate_dma_handle().
size Size of buffer to be released.
iova I/O virtual address of the shared memory.
vaddr Host virtual address of the shared memory.
shared_mem_attr Bit mask that was used to allocate the shared memory.

DESCRIPTION

The wsio_free_shared_mem() WSIO function is called by a device driver to release an I/O virtually contiguous DMA buffer that was allocated by wsio_allocate_shared_mem(). All parameters to the macro should be the same as those passed to the call that allocated the memory.

wsio_free_shared_mem() can be called in a non-blocking context.

RETURN VALUES

None
CONSTRAINTS

EXAMPLE

```c
void *dma_handle = NULL;
wsio_iova_t io_virtual_addr;
wsio_vaddr_t host_virtual_addr;

dma_handle = wsio_allocate_dma_handle(isc_entry);
if (dma_handle == NULL) {
    /* No handle allocated. */
    return(ERROR);
}
/* The DMA handle is now in the dma_handle variable */

if (wsio_allocate_shared_mem(isc_entry,dma_handle,buf_size,
        &io_virtual_addr,
        &host_virtual_addr,0)
        != WSIO_MAP_OK) {
        /* Unable to allocate shared memory,
           * so return an error */
        return(ERROR);
}
/* A buffer of size 'buf_size' is now allocated and mapped
   * in both host virtual memory and I/O virtual memory space.
   */
wsio_free_shared_mem(isc_entry,dma_handle,buf_size,
        io_virtual_addr,host_virtual_addr,0);
```

SEE ALSO

- wsio_allocate_dma_handle(WSIO3),
- wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
- wsio_free_dma_handle(WSIO3), wsio_flush_shared_mem(WSIO3),
- wsio_iova_to_phys(WSIO3), wsio_set_device_attributes(WSIO3),
- wsio_set_dma_attributes(WSIO3)
NAME

wsio_generic_event_t(WSIO5) – Generic WSIO event information

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This is a generic WSIO event information data structure. Any event in the system which is of relevance to a driver is managed through this data structure.

STRUCTURE MEMBERS

struct wsio_generic {
    wsio_event_t event;
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;

The fields in the structure are:

event              A structure of type wsio_event_t, which indicates the event.
event_id           A WSIO provided event_id.
isc                Pointer to the isc(struct isc_table_type) structure for the instance of the driver.
wsio_completion_cb A WSIO provided completion callback routine. Once the driver processes the event, this is the completion call back to WSIO.
arg                An event related argument. This could be used in the context of the event and might provide further information relevant to the event.
EXAMPLE

If the driver has a transaction based interrupt, the driver is notified if that interrupt is being reassigned to a new CPU. The *wsio_generic_event_t* structure is filled with the following information:

- `event` : `WSIO_EVENT_OFFLINE_CPU`
- `event_id` : WSIO provided
- `isc` : Pointer to the instance of the associated driver *isc_table_type* structure.
- `wsio_completion_cb` : WSIO provided call back.
- `arg` : Pointer to a structure of type *wsio_intr_migt_t*

If a driver is using a transaction based interrupt, the driver must register for the `WSIO_EVENT_OFFLINE_CPU` event. Refer to the Interrupt Migration chapter in the *Driver Development Guide* for related information.

SEE ALSO

- `wsio_drv_event_t(WSIO5)`
- `wsio_install_drv_event_handler(WSIO3)`
- `wsio_reg_drv_capability_mask(WSIO5)`
NAME

```
wsio_get_active_processor_count(WSIO3) – Get the number of active CPUs in the system.
```

SYNOPSIS

```
#include <wsio/wsio.h>

int wsio_get_active_processors_count(void);
```

PARAMETERS

DESCRIPTION

The `wsio_get_active_processor_count()` WSIO function returns the number of CPUs currently active. On a running system the active CPU number may change. For example, if a processor is deallocated or allocated.

RETURN VALUES

The number of active processors.

CONSTRAINTS

EXAMPLE

```
int active_cpus;

active_cpus = wsio_get_active_processor_count();
printf("number of active CPUs %d\n", active_cpus);
```

SEE ALSO

```
wsio_get_processor_count(WSIO3)
```
NAME

wsio_get_all_registers(WSIO3) – Get an array of all available device registers

SYNOPSIS

#include <wsio/wsio.h>

wsio_reg_info_t * wsio_get_all_registers (struct isc_table_type * isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The wsio_get_all_registers() WSIO function obtains an array of all of the registers for the device associated with the isc_table_type entry that is passed in. This function assumes that the device driver will know how many registers will be returned in the array and what their uses are.

RETURN VALUES

This function returns the address of an array of structures of type wsio_reg_info_t. If no registers exist, NULL will be returned.

CONSTRAINTS
EXAMPLE

```c
wsio_reg_info_t *registers;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error. */
    return(ERROR);
}

/* All of the devices registers are now contained in the
 * 'registers' variable. They may be mapped as follows:
 */
if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will
 * now be mapped.
 */
```

SEE ALSO

wsio_map_reg(WSIO3), wsio_read_regXX(WSIO3),
wsio_unmap_reg(WSIO3), wsio_write_regXX(WSIO3)
NAME

*wsio_get_ioports*(WSIO3) – Obtain the addresses and sizes of I/O ports.

SYNOPSIS

```
#include <wsio/wsio.h>

int wsio_get_ioports (struct isc_table_type *isc, 
                      int cnt, wsio_iop_t port_array[]);
```

PARAMETERS

- **isc** Pointer to the driver’s `isc_table` entry.
- **cnt** Indicates the maximum number of ports to return.
- **port_array** Array to store the ports in.

DESCRIPTION

The `wsio_get_ioports()` WSIO function gets up to `cnt` I/O ports and stores them in an array. All ports will be obtained if the `cnt` variable is large enough. The driver must allocate enough space for all ports inside the `port_array`. Once these ports have been obtained, they can be mapped using `wsio_map_port()` and accessed using `wsio_port_inXX()`, and `wsio_port_outXX()`.

RETURN VALUES

- **WSIO_OK** Successful completion.
- **WSIO_ERROR** There was a parameter error.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t ioports_array[10];
/* An array with enough space for all ports needs to * be allocated */

if (wsio_get_ioports(isc_entry,10,ioports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,ioports_array[0].addr,
                 ioports_array[0].size, &port_handle
                 ) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now unmap the port */
if (wsio_unmap_port(isc_entry,ioports_array[0].addr,
                    ioports_array[0].size, port_handle
                   ) != WSIO_OK) {
    /* There was an error unmapping the port */
    return(ERROR);
}
```

SEE ALSO

wsio_map_port (WSIO3), wsio_port_inXX (WSIO3),
wsio_port_outXX (WSIO3)
NAME

wsio_get_isc(WSIO3) – Get the ISC structure pointer for a device file.

SYNOPSIS

```
#include <sys/io.h>
#include <wsio/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**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Successful completion. The ISC pointer found is returned in <code>isc_ptr</code>.</td>
</tr>
<tr>
<td>0</td>
<td>Failure. The ISC could not be found.</td>
</tr>
</tbody>
</table>

**CONSTRAINTS**

**SEE ALSO**
NAME

wsio_get_processor_count(WSIO3) – Get the number of CPUs in the system.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_get_processor_count(void)

PARAMETERS

DESCRIPTION

The wsio_get_processor_count() WSIO function returns the number of CPUs configured into the system. On a running system the number of CPUs may change. For example, if a processor is deallocated or allocated.

RETURN VALUES

The number of processors.

CONSTRAINTS

EXAMPLE

int num_cpus;

num_cpus = wsio_get_processor_count();

printf("number of CPUs %d\n", num_cpus);

SEE ALSO

wsio_get_active_processor_count(WSIO3)
NAME

wsio_get_system_params(WSIO3) – Get information about the system.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_get_system_params_t * wsio_get_system_params (
        struct isc_table_type * isc, wsio_get_sys_parm_t id,
        wsio_unintptr_t * parm);

PARAMETERS

isc Pointer to the driver's isc_table entry.
id An identifier indicating what parameter to get.
parm Value returned will be written to a memory location pointed to by parm.

id parm

WSIO_CACHELINE_SIZE Indicates the cacheline size in bytes
WSIO_DEFAULT_PAGE_SIZE Indicates the default page size in bytes.
WSIO_DMA_64BIT_ADDRESSING Indicates 64-bit addressing capability (1), or not capable (0).
WSIO_DMA_COHERENT_IO Indicates IO coherent (1), or not coherent (0).
WSIO_DMA_IOPDIR_PRESENT Indicates IOPDIR is present (1) or not present (0).
WSIO_NUM_CPUS Indicates the number of CPUs on the system.
DESCRIPTION

The `wsio_get_system_params()` WSIO function obtains all system parameters that are currently defined. The `id` is to identify what parameter to retrieve. The value that the caller is interested in will be stored into the memory location pointed to by `parm`.

RETURN VALUES

- **WSIO_OK**: Successful completion.
- **WSIO_ERROR**: Error.

CONSTRAINTS

EXAMPLE

```c
int ret;

wsio_uintptr_t value;

ret = wsio_get_system_params(isc_entry,
                            WSIO_CACHELINE_SIZE, &value);

/* The cacheline size will be obtained and put into value */
```

SEE ALSO
NAME

`wsio_init_map_context` (WSIO3) – Initialize the context used for DMA mapping.

SYNOPSIS

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

void wsio_init_map_context (wsio_map_context_t * context);
```

PARAMETERS

- `context` Pointer to the context to be initialized.

DESCRIPTION

The `wsio_init_map_context()` WSIO macro is called by device drivers to initialize a `context` that is needed by `wsio_map_dma_buffer()`, and other map related functions. A `context` is used internally by mapping services so that system resources can be used efficiently. The same `context` should be used for a set of DMA mappings that are all going to be unmapped at the same time. This `context` should be initialized before its first use. If the mapping that is being performed is independent of all other mappings, then a `context` is not necessary and `NULL` should be passed to the mapping function in the context field.

RETURN VALUES

None

CONSTRAINTS
EXAMPLE

```c
wsio_map_context_t dma_context;

wsio_init_map_context(&dma_context);
/* dma_context is now ready to be used for mapping. It can
 * be used for any number of mappings as long as they are
 * all unmapped at the same time
 */
```

SEE ALSO

`wsio_allocate_dma_handle(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_iova_to_phys(WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3).`
NAME

`wsio_install_driver` (WSIO3) – Install a driver's header structure into the WSIO CDIO.

SYNOPSIS

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

int wsio_install_driver (void * 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

- 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
/* 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
};
wsio_install_driver(WSIO3)

/* 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_install_drv_event_handler (WSIO) – Install a driver's event handler

SYNOPSIS

#include <wsio/wsio.h>

int wsio_install_drv_event_handler (wsio_drv_info_t * drv_info,
    void (* drv_handler)(wsio_generic_event_t * generic_ptr);

PARAMETERS

drv_info Pointer to the driver's wsio_drv_info_t structure.
drv_handler Function pointer to a driver's event handler

DESCRIPTION

The wsio_install_drv_event_handler() WSIO function is called by a
device driver to register an event handler to deal with events. When an
event occurs, such as a suspend request, WSIO will invoke this handler.
This function should be called in a driver's installation routine after
wsio_install_driver() is executed.

RETURN VALUES

WSIO_OK Returned on success.

WSIO_DRV_NOT_FOUND An earlier call to wsio_install_driver() was not
    successful.

WSIO_HANDLER_NULL Drv_handler is a NULL pointer.

WSIO_INFO_NULL Drv_info is a NULL pointer.

CONSTRAINTS
EXAMPLE

```c
static wsio_drv_info_t my_drv_info {
    &my_info,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION,
    /* This is a macro defined in wsio.h, bearing a version
     * stamp */
}

my_driver_install() { 
    if (wsio_install_driver(&my_drv_info) != 1) { 
        return (0); /* 0 means error */
    }
    /* Register the event handler with WSIO */
    if (wsio_install_drv_event_handler(
            &my_drv_info, my_handler) != WSIO_OK) {
        /* The example given shows that wsio_uninstall_driver() 
         * is called. In this case, the driver 
         * will not be brought up. If a driver chooses to 
         * ignore the error and proceed, it will 
         * run without being able to handle events */
        wsio_uninstall_driver(&my_drv_info);
        return (0); /* 0 means error */
    }
}
```

SEE ALSO

wsio_query_supported_function(WSIO_DRV),
wsio_reg_drv_capability_mask(WSIO5),
wsio_uninstall_drv_event_handler(WSIO3)
NAME

\texttt{wsio\_intr\_activate(WSIO5)} – Enable an interrupt object.

SYNOPSIS

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

int wsio_intr_activate (struct isc_table_type *isc,
                        wsio_intr_object_t obj);
\end{verbatim}

PARAMETERS

\begin{itemize}
  \item \textit{isc} \hspace{1cm} Pointer to the driver's isc_table entry.
  \item \textit{obj} \hspace{1cm} Interrupt object to enable.
\end{itemize}

DESCRIPTION

The \texttt{wsio\_intr\_activate()} WSIO function activates an interrupt object that was allocated with \texttt{wsio\_intr\_alloc()}. The interrupt object must be activated before the system will call the device driver's ISR (as specified in \texttt{wsio\_intr\_alloc()}). It is assumed that (if possible) the device will not generate interrupts until after this function is called. The \texttt{wsio\_intr\_deactivate()} or \texttt{wsio\_intr\_deactivate\_nowait()} service undoes the effects of this function.

Attempting to activate an interrupt object that is already active is an error condition that returns \texttt{WSIO\_INTR\_ACTIVATED}, without modifying the interrupt object.

If interrupt migration software is present in the system, the following scenarios relate to a \texttt{wsio\_intr\_activate()} caller.

If a driver invokes this routine as part of a non-WSIO event to activate a line based card interrupt (for instance, a card reset) and if interrupt migration is in progress, the invocation fails with a \texttt{WSIO\_ERROR}.

When using transaction based interrupts as part of interrupt migration operation, the drivers must invoke \texttt{wsio\_intr\_activate()}, \texttt{wsio\_intr\_set\_cpu\_spec()}, and \texttt{wsio\_intr\_deactivate()} routines. Therefore, the driver must take care of the synchronization of any two driver threads invoking the routines at the same time.
See the Interrupt Migration chapter of the *Driver Development Guide* for related information.

**RETURN VALUES**

WSIO_OK  
Operation succeeded.

WSIO_ERROR  
Failure; no interrupt services available or interrupt migration might be in progress.

WSIO_INTR_INV_OBJ  
Must call wsio_intr_set_cpu_spec() or wsio_intr_set_irq_line() first.

WSIO_INTR_ACTIVATED  
obj already active.

WSIO_PARM_ERROR  
Invalid parameters.

**CONSTRAINTS**

**EXAMPLE**

```c
/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                       (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
                                WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
```
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_alloc(WSIO3), wsio_intr_deactivate(WSIO5),
wsio_intr_deactivate_nowait(WSIO3),
wsio_intr_set_cpu_spec(WSIO3), wsio_intr_set_irq_line(WSIO3)
NAME

`wsio_intr_alloc(WSIO3)` – Allocate an interrupt object.

SYNOPSIS

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

int wsio_intr_alloc (struct isc_table_type * isc,
                     wsio_drv_isr_t isr,
                     uintptr_t arg,
                     uint64_t flags,
                     wsio_intr_object_t * obj);
```

PARAMETERS

- **isc**  Pointer to the driver’s `isc_table` entry.
- **isr**  Address of the interrupt service routine.
- **arg**  Argument to be passed to the `driver_isr`.
- **flags**  Shared or exclusive flag.
- **obj**  Interrupt object.

DESCRIPTION

The `wsio_intr_alloc()` function allocates and initializes an interrupt object that will field interrupts generated by the given device associated with the `isc`. This routine may sleep and thus must be called from a thread context. This routine might allocate hardware resource, so it should be used with care.

The `flags` parameter should be either `WSIO_INTR_EXCLUSIVE` if the device driver’s ISR cannot be shared, or zero if the ISR can be shared. If the `WSIO_INTR_EXCLUSIVE` flag is not present, the driver’s ISR may be called even if the device did not generate an interrupt. If the device driver has no way of determining if the card needs servicing, the device driver’s ISR cannot be shared. For such a card, running the device driver’s ISR indicates the card needs servicing.

The function may block, so it must be called in the thread context.
RETURN VALUES

WSIO_OK        Successful completion.
WSIO_ERROR     Failure; no interrupt services available.
WSIO_INTR_INV_FLAG
    Must specify 0 (shared) or WSIO_INTR_EXCLUSIVE only.
WSIO_MEM_ALLOC_FAILED
    Interrupt services failed to allocate memory.
WSIO_PARM_ERROR
    Invalid parameters

CONSTRAINTS

EXAMPLE

/* Allocate a line based interrupt */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object for a shared interrupt */
status = wsio_intr_alloc(isc, isr,
                         (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_free(WSIO3)
NAME

wsio_intr_deactivate(WSIO5) – Disable an interrupt object.

SYNOPSIS

```
#include <wsio/wsio.h>

int wsio_intr_deactivate (struct isc_table_type *isc,
                           wsio_intr_object_t obj);
```

PARAMETERS

isc Pointer to the driver's isc_table entry.

obj Interrupt object.

DESCRIPTION

The wsio_intr_deactivate() WSIO function deactivates an interrupt object that was previously activated with wsio_intr_activate(). By deactivating the interrupt object the system will stop calling the device driver's ISR (as specified in wsio_intr_alloc()). It is assumed that (if possible) the device's interrupts will be disabled before this function is called. This function will sleep.

Attempting to deactivate an interrupt object that has not been activated is an error condition that returns WSIO_INTR_DEACTIVATED, without modifying the interrupt object.

If interrupt migration software is present in the system, the following scenarios are relevant to a wsio_intr_deactivate() caller.

If a driver invokes this routine as part of a non-WSIO event to activate a line based card interrupt (such as a card reset) and if interrupt migration operation is in progress, the invocation fails with a WSIO_ERROR.

When drivers use transaction based interrupts as part of the interrupt migration operation, they must invoke wsio_intr_activate(), wsio_intr_set_cpu_spec() and wsio_intr_deactivate() routines. Therefore, the driver must take care of the synchronization of any two driver threads invoking the routines at the same time.
See the Interrupt Migration chapter of the *Driver Development Guide* for related information.

**RETURN VALUES**

- **WSIO_OK**  
  Successful completion.

- **WSIO_INTR_DEACTIVATED**  
  obj not active.

- **WSIO_PARM_ERROR**  
  Invalid parameters.

- **WSIO_ERROR**  
  Failure; no interrupt services available or interrupt migration might be in progress.

**CONSTRAINTS**

Must not be called in an interrupt context.

**EXAMPLE**

```c
/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,(uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj, 
        WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}
```
/* deactivate the interrupt */
status = wsio_intr_deactivate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_deactivate(WSIO5),
sio_intr_deactivate_nowait(WSIO3),
wsio_intr_set_cpu_spec(WSIO3), wsio_intr_set_irq_line(WSIO3),
NAME

wsio_intr_deactivate_nowait(WSIO3) – Disable an interrupt object with callback.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_deactivate_nowait (struct isc_table_type *isc,
                                 wsio_intr_object_t obj,
                                 wsio_intr_deact_cb_t callback_func,
                                 uintptr_t arg);

PARAMETERS

isc Pointer to the driver's isc_table entry.
obj Interrupt object.
callback_func A driver callback function that will be executed.
arg Parameter passed back to a driver in the callback.

DESCRIPTION

The wsio_intr_deactivate_nowait() WSIO function is a mirror of
wsio_intr_deactivate() except that this one does not sleep. It
deactivates an interrupt object that was previously activated with
wsio_intr_activate(). By deactivating the interrupt object the system
will stop calling the device driver's ISR (as specified in
wsio_intr_alloc()). It is assumed that, if possible, the device's
interrupts will be disabled before this function is called.

Attempting to deactivate an interrupt object that has not been activated
is an error condition that returns WSIO_INTR_DEACTIVATED, without
modifying the interrupt object.

This function will return immediately. Upon receiving confirmation that
the interrupts have been disabled, the callback_func will be executed
passing the arg back to the driver. Since this call will not sleep, it can be
called in a non-blocking context.
RETURN VALUES

- **WSIO_OK**: Successful completion.
- **WSIO_INTR_DEACTIVATED**: `obj` not active.
- **WSIO_ERROR**: Operation failed.
- **WSIO_INTR_INV_OBJ**: Must call `wsio_intr_set_cpu_spec()` or `wsio_intr_set_irq_line()` first.
- **WSIO_NO_INTR_CB**: No callback function passed in.
- **WSIO_PARM_ERROR**: Invalid parameters.

CONSTRAINTS

EXAMPLE

```c
/* Driver’s callback function for
 wsio_intr_deactivate_nowait() */
void my_callback(struct isc_table_type *isc)
{
    ........
}

/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
    (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
    WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
```
return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

............
/* deactivate the interrupt */
status = wsio_intr_deactivate_nowait(isc, obj, my_callback, isc);
if (status != WSIO_OK) {
    return(ERROR);
}

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_deactivate(WSIO5), wsio_intr_set_cpu_spec(WSIO3),
wsio_intr_set_line(WSIO3)
NAME

wsio_intr_free(WSIO3) – Free an interrupt object.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_free (struct isc_table_type *isc, wsio_intr_object_t obj);

PARAMETERS

isc Pointer to the driver's isc_table entry.
obj Interrupt object.

DESCRIPTION

Before this function returns, the services guarantee that all outstanding ISRs have been run to completion. The obj object must not be active (either never activated, or deactivated with wsio_intr_deactivate(), or wsio_intr_deactivate_nowait()) when this function is called.

This function may block, so it must be called in a thread context.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_INTR_ACTIVATED

Must call wsio_intr_deactivate or wsio_intr_deactivate_nowait first.
WSIO_PARM_ERROR
Invalid parameters.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                        (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}
/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
                                WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}
/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}
/* deactivate the interrupt */
status = wsio_intr_deactivate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}
/* free the interrupt obj */
status = wsio_intr_free(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}
obj = NULL;

SEE ALSO

wsio_intr_alloc(WSIO3)
NAME

wsio_intr_get_assigned_cpu(WSIO3) – Get the currently assigned CPU for the interrupt object.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_assigned_cpu (wsio_intr obj, intptr_t * cpu_spec);

PARAMETERS

obj Interrupt object.

cpu_spec CPU specification.

DESCRIPTION

The wsio_intr_get_assigned_cpu() WSIO function returns the currently assigned CPU for the passed interrupt object. The return type for cpu_spec will depend on platform. On PA, it will return assigned CPU in the location pointed by cpu_spec.

INTR_ATTR_ASSIGNED_CPU attribute is not currently defined in BN-CDIO spec, but will be added as it is a necessary functionality.

RETURN VALUES

WSIO_OK Successful completion.

WSIO_ERROR INTR_ATTR_ASSIGNED_CPU not implemented in CDIO.

WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS
EXAMPLE

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
uintptr_t cpu_spec;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
    (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a TXN based interrupt */
status = wsio_intr_set_cpu_spec(isc, obj, WSIO_INTR_CPU_ANY);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the CPU spec */
status = wsio_intr_get_assigned_cpu(obj, &cpu_spec);
if (status != WSIO_OK) {
    return(ERROR);
}

printf("assigned cpu for the interrupt object is = %d\n",
    cpu_spec);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_cpu_spec(WSIO3)
NAME

wsio_intr_get_irq_line(WSIO3) – Get the interrupt line number.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_get_irq_line (struct isc_table_type *isc,
                          wsio_intr_object_t obj,
                          intptr_t *irq_line_num);

PARAMETERS

isc Pointer to the driver's isc_table entry.
obj Interrupt object.
irq_line_num The interrupt line number.

DESCRIPTION

The wsio_intr_get_irq_line() WSIO function returns the line number that the given obj is currently using. In most cases this routine is not necessary, as WSIO_IRQ_LINE_AUTO, can be given to wsio_intr_set_irq_line(). However, if for some reason a device driver needs to know the interrupt line that a card should use, this function provides the necessary data. On a running system the line number assigned to an obj may change. For example, when a processor is deallocated, the driver services will reassign the interrupt line on behalf of the driver.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_INTR_INV_OBJ Must call wsio_intr_set_irq_line() first.
WSIO_INTR_ACTIVATED Object not active, call wsio_intr_activate() first.
WSIO_ERROR Failed to get line number.
 CONSTRAINTS

EXAMPLE

/* Allocate a line based interrupt and activate it */
wsio_intr_object_t obj;
int status;
intptr_t irq;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
    (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio_intr_set_irq_line(isc, obj,
    WSIO_IRQ_LINE_AUTO, 0);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the line number */
status = wsio_intr_get_irq_line(isc, obj, &irq);
if (status != WSIO_OK) {
    return(ERROR);
}

printf ("my line number is %ld\n", irq);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_line(WSIO3)
NAME

wsio_intr_get_txn_info(WSIO3) – Get the transaction address and data value.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_get_txn_info (struct isc_table_type *isc,
                          wsio_intr_object_t obj,
                          intptr_t *txn_addr,
                          intptr_t *txn_data);

PARAMETERS

isc Pointer to the driver's isc_table entry.
obj Interrupt object.
txn_addr Transaction address value.
txn_data Transaction data value.

DESCRIPTION

The wsio_intr_get_txn_info() WSIO function returns the transaction address and transaction data associated with a transaction based obj.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_INTR_INV_OBJ Must be a transaction based obj; call wsio_intr_set_cpu_spec() first.
WSIO_ERROR Failed to get values.
WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS
EXAMPLE

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
intptr_t txn_addr, txn_data;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                          (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a TXN based interrupt */
status = wsio_intr_set_cpu_spec(isc, obj, WSIO_INTR_CPU_ANY);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the TXN values */
status = wsio_intr_get_txn_info(isc, obj,
                                &txn_addr, &txn_data);
if (status != WSIO_OK) {
    return(ERROR);
}

printf("txn_addr = %ld\n", txn_addr);
printf("txn_data = %ld\n", txn_data);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_cpu_spec(WSIO3)
NAME

\texttt{wsio\_intr\_migr\_t(WSIO5)} – Driver - WSIO communication structure during interrupt migration.

SYNOPSIS

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

DESCRIPTION

This structure is used for communication between drivers and WSIO when interrupt migration is being processed.

See the Interrupt Migration chapter in the \textit{Driver Development Guide} for relevant information.

STRUCTURE MEMBERS

Table 4-3 Driver Relevant \texttt{wsio\_intr\_migr\_t} Structure Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{wsio_intr_object_t}</td>
<td>\texttt{intr_obj}</td>
</tr>
<tr>
<td>\texttt{intptr_t}</td>
<td>\texttt{dest_spu}</td>
</tr>
<tr>
<td>\texttt{wsio_intr_migr_info_t}</td>
<td>\texttt{migr_info}</td>
</tr>
<tr>
<td>\texttt{wsio_ret_code_t}</td>
<td>\texttt{ret_val}</td>
</tr>
<tr>
<td>\texttt{void*}</td>
<td>\texttt{resvd}</td>
</tr>
</tbody>
</table>

\texttt{intr\_obj} Interrupt object of the interrupt being moved.

\texttt{dest\_spu} CPU ID of the CPU to which the interrupt is to be moved.
The value of dest_spu depends on the event and can be as indicated in the following table:

Table 4-4 dest_spu values

<table>
<thead>
<tr>
<th>dest_spu</th>
<th>migr_info</th>
<th>Event</th>
<th>Descr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>WSIO_LBI_INTR_MIG_NOTIFY</td>
<td>WSIO_LBI_INTR_MIG</td>
<td>The notify event is sent to all LBI drivers which have registered for the WSIO_LBI_INTR_MIG event. dest_spu is not valid here.</td>
</tr>
<tr>
<td>spu_id</td>
<td>WSIO_LBI_INTR_MIG_COMPLETE</td>
<td>WSIO_LBI_INTR_MIG</td>
<td>The LBI drivers, which have registered for the WSIO_LBI_INTR_MIG event are notified after the interrupt migration has completed. The dest_spu is the “new” CPU to which the interrupt has migrated.</td>
</tr>
<tr>
<td>-1/spu_id</td>
<td>N/A</td>
<td>WSIO_OFF LINE_CPU</td>
<td>If the dest_cpu is not -1, the spu_id should be used by drivers in the wsio_intr_set_cpu_spec() call to migrate the interrupt to this CPU. If it is -1, drivers can pass in WSIO_INTR_CPU_ANY, WSIO_INTR_CPU_ANY_UNIQUE, or a spu_id (see wsio_intr_set_cpu_spec()).</td>
</tr>
</tbody>
</table>

migr_info More information about the migration event; see wsio_intr_migr_info_t.
ret_val Return value of migration.
resvd Reserved field.
SEE ALSO

wsio_intr_migr_info(WSIO5), wsio_drv_event_t(WSIO5),
wsio_intr_set_cpu_spec(WSIO3)
NAME

wsio_intr_migr_info_t(WSIO5) – Event notification for migration of line based interrupts

SYNOPSIS

#include <wsio/wsio.h>

DESCRIPTION

This enumerator is used to notify the LBI drivers, which have registered with WSIO, about an interrupt migration event. The wsio_intr_migr_t structure is used for communication between WSIO and the driver. This structure is not utilized with TBI drivers.

Refer to the Interrupt Migration chapter of the Driver Development Guide for relevant information.

STRUCTURE MEMBERS

typedef enum wsio_intr_migr_info {
    WSIO_LBI_INTR_MIGR_NOTIFY = 1,
    WSIO_LBI_INTR_MIGR_COMPLETE
} wsio_intr_migr_info_t;

WSIO_LBI_INTR_MIGR_NOTIFY  WSIO sends a notification to the LBI driver that the interrupt will be migrated.

WSIO_LBI_INTR_MIGR_COMPLETE  WSIO notifies the driver of completion of the interrupt migration. The spu_id field of wsio_intr_migr_t is set to cpu_id of the CPU to which the interrupt has been migrated.

SEE ALSO

wsio_intr_migr_t(WSIO5), wsio_drv_event_t(WSIO5),
wsi0_intr_set_cpu_spec(WSIO3)
NAME

wsio_intr_set_cpu_spec(WSIO3) – Initialize and distribute transaction based interrupts.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_intr_set_cpu_spec (struct isc_table_type *isc,
       wsio_intr_object_t obj,
       intptr_t cpu_spec);

PARAMETERS

isc Pointer to the driver's isc_table entry.
obj Interrupt object.
cpu_spec CPU load balancing specification. If cpu_spec is not one of the two flags described below but is a txn_addr, the interrupt will be bound to this CPU.

<table>
<thead>
<tr>
<th>cpu_spec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_INTR_CPU_ANY</td>
<td>The services will select any processor. The device driver does not care which processor it is bound to.</td>
</tr>
<tr>
<td>WSIO_INTR_CPU_ANY_UNIQUE</td>
<td>The services will select a processor that this adapter card does not already send interrupts to; this is not currently implemented. The services will return a WSIO_NOT_IMPLEMENTED error if the driver specifies WSIO_INTR_CPU_ANY_UNIQUE for cpu_spec.</td>
</tr>
<tr>
<td>txn_addr</td>
<td>Bind to this CPU address. If a processor number is specified, it should be a value between zero and the number of CPUs minus one.</td>
</tr>
</tbody>
</table>
DESCRIPTION

The `wsio_intr_set_cpu_spec()` WSIO function is used to initialize and distribute transaction based interrupts. All drivers using transaction based interrupts must register to the `WSIO_EVENT_OFFLINE_CPU` before calling this function. The drivers can register for this mandatory event through the `wsio_regDrvCapabilityMask()` call. The registration routine is invoked in the attach routine after the completion of `isc_claim()` The Interrupt Migration chapter of the Driver Development Guide has relevant information.

The function can block and should be called in the thread context. If the function is called multiple times each successive call overrides the value of the previous call.

RETURN VALUES

- **WSIO_OK**: Successful completion.
- **WSIO_INTR_INV_OBJ**: Must be a transaction based `obj`; call `wsio_intr_set_cpu_spec()` first.
- **WSIO_ERROR**: Failed to set `cpu_spec`.
- **WSIO_EXCLUSIVE_FAILED**: Can not get an exclusive interrupt.
- **WSIO_INTR_ACTIVATED**: `obj` is currently active; must call `wsio_intr_deactivate()` first.
- **WSIO_NOT_IMPLEMENTED**: `cpu_spec` algorithm not implemented.
- **WSIO_INTR_INV_CPU_NUM**: Invalid CPU number specified.
- **WSIO_PARM_ERROR**: Invalid parameters.

CONSTRAINTS
EXAMPLE

The following is an example of a driver registering for a
WSIO_EVENT_OFFLINE_CPU event:

```c
driver_attach(...) {
    wsio_event_mask_t newmask;
    .
    .
    isc_claim(isc);
    newmask = oldmask | WSIO_EVENT_OFFLINE_CPU;
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    .
    .
}
```

The following is an example of a driver calling
wsio_intr_set_cpu_spec().

```c
/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
intptr_t txn_addr, txn_data;

/* allocate an interrupt object */
status = wsio_intr_alloc(isc, driver_isr,
                        (uintptr_t)isc, 0, &obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* Get a TXN based interrupt */
status = wsio_intr_set_cpu_spec(isc, obj, WSIO_INTR_CPU_ANY);
if (status != WSIO_OK) {
    return(ERROR);
}

/* activate the interrupt */
status = wsio_intr_activate(isc, obj);
if (status != WSIO_OK) {
    return(ERROR);
}

/* get the TXN values */
status = wsio_intr_get_txn_info(isc, obj,
                                &txn_addr, &txn_data);
if (status != WSIO_OK) {
    return(ERROR);
}
```
printf ("txn_addr = %ld\n", txn_addr);
printf ("txn_data = %ld\n", txn_data);

SEE ALSO

wsio_intr_activate(WSIO5), wsio_intr_alloc(WSIO3),
wsio_intr_set_cpu_spec(WSIO3),
wsio_intr_get_assigned_cpu(WSIO3)
**NAME**

`wsio_intr_set_irq_line(WSIO3)` – Set the interrupt line number.

**SYNOPSIS**

```
#include <wsio/wsio.h>

int wsio_intr_set_irq_line (struct isc_table_type *isc,
                         wsio_intr_object_t obj,
                         intptr_t *irq_line_num,
                         uint64_t flags);
```

**PARAMETERS**

- `isc` | Pointer to the driver’s isc_table entry.
- `obj` | Interrupt object.
- `irq_line_num` | The interrupt line number, or `WSIO_IO_INT_LINE_AUTO`.
- `flags` | Zero (level sensitive) or `WSIO_INTR_EDGE_SENSITIVE`.

**DESCRIPTION**

The `wsio_intr_set_irq_line()` WSIO function is used to setup a line based `obj`. The `WSIO_INTR_ACTIVATED` error code will be returned if this function is called on an active interrupt object. Most drivers will use `WSIO_IRQ_LINE_AUTO` for the value of the `irq_line_num` parameter which forces the services to determine the interrupt line value for the particular device.

If the device generates level-sensitive interrupts, the `flags` parameter should be zero. If the device generates edge-sensitive interrupts, the `flags` parameter should be `WSIO_INTR_EDGE_SENSITIVE`.

Calling this interface multiple times will not move this interrupt from one processor to another in a round-robin manner. Refer to the Interrupt Migration chapter of the *Driver Development Guide* for driver related changes and impact.
RETURN VALUES

WSIO_OK Operation succeeded.

WSIO_INTR_INV_OBJ Must not be a transaction based interrupt.

WSIO_INTR_ACTIVATED Object is active; do not call \texttt{wsio\_intr\_activate()} first.

WSIO_ERROR Failed to set line number.

WSIO_INTR_INV_FLAG Must be zero (level) or \texttt{WSIO\_INTR\_EDGE\_SENSITIVE}.

WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

EXAMPLE

/* Allocate a line based interrupt */
wsio\_intr\_object\_t obj;
int status;

/* allocate an interrupt object */
status = wsio\_intr\_alloc(isc, driver\_ isr,
    (uintptr\_t)isc, 0, &obj);
if (status != WSIO\_OK) {
    return(ERROR);
}

/* Get a Level Sensitive IRQ */
status = wsio\_intr\_set\_irq\_line(isc, obj,
    WSIO\_IRQ\_LINE\_AUTO, 0);
if (status != WSIO\_OK) {
    return(ERROR);
}

SEE ALSO

\texttt{wsio\_intr\_activate(WSIO5)}, \texttt{wsio\_intr\_alloc(WSIO3)},
\texttt{wsio\_intr\_set\_line(WSIO3)}
NAME

wsio_io_sync(WSIO3) – Perform a sync of shared memory if necessary.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_io_sync (struct isc_table_type * isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The wsio_io_sync() WSIO function synchronizes a device's view and the host's view of memory. This functionality can also be achieved by doing a PIO read of a card register, but in some implementations this function may provide a lower latency mechanism.

wsio_io_sync() can be called in a non-blocking context.

RETURN VALUES

None

CONSTRAINTS

EXAMPLE

wsio_io_sync(isc_entry);

SEE ALSO

WSIO_ORDERED_INTERRUPTS(WSIO3)
NAME

wsio_iova_to_phys (WSIO3) – Translate an I/O virtual address to a physical address.

SYNOPSIS

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

caddr_t wsio_iova_to_phys (struct isc_table_type *isc, 
    void *dma_handle, wsio_iova_t iova);
```

PARAMETERS

- `isc` Pointer to the driver’s `isc_table` entry.
- `dma_handle` DMA handle allocated using `wsio_allocate_dma_handle()`.
- `iova` I/O virtual address to be translated.

DESCRIPTION

The `wsio_iova_to_phys()` WSIO function is called by a device driver to translate an I/O virtual address to a physical address.

`wsio_iova_to_phys()` can be called in a non-blocking context.

RETURN VALUES

The physical address corresponding to `iova` or NULL if a translation does not exist.

CONSTRAINTS
EXAMPLE

caddr_t phys_address = NULL;

phys_address =
wsio_iova_to_phys(isc_entry,dma_handle,io_address);

/* The 'phys_address' variable will now contain the physical
 * address that 'iova' translates to. */

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context (WSIO3), wsio_map_dma_buffer (WSIO3),
wsio_remap_dma_buffer (WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_unmap_dma_buffer (WSIO3).
NAME

wsio_isc_to_instance(WSIO3) – Retrieve an instance number of an iotree node

SYNOPSIS

```
#include <wsio/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. SCSI interface drivers must check the instance number returned by `wsio_isc_to_instance()`. If the returned value is greater than `SCSI_MAX_BUS_ID`, the driver should return `WSIO_ERROR`.

- 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

>=0  Successful completion. The value is the matching instance number.

-1   Error.

CONSTRAINTS

SEE ALSO
NAME

WSIO_LITTLE_ENDIAN(WSIO3) – Macro to return true (1) if the local bus is little-endian.

SYNOPSIS

#include <wsio/wsio.h>

int WSIO_LITTLE_ENDIAN(struct isc_table_type * isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The WSIO_LITTLE_ENDIAN() macro is called by a device driver to report whether the local bus is little-endian. If it is, true is returned; otherwise it returns false. This can be used by a driver along with the known endianness of the host processor to decide whether endian swapping should be performed. Endian swapping might be necessary for any data transfers between the I/O bus and local host memory.

WSIO_LITTLE_ENDIAN() can be called in a non-blocking context.

RETURN VALUES

Returns a one (1) if the local bus is little-endian, and a zero (0) otherwise.

CONSTRAINTS

EXAMPLE

if (WSIO_LITTLE_ENDIAN(isc_entry)) {
   /* Endian swapping must be performed
} else {
   /* No endian swapping necessary */
}
SEE ALSO

WSIO_BIG_ENDIAN(WSIO3)
NAME

wsio_map_cfg_handle(WSIO3) – Obtain a configuration space access handle.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_map_cfg_handle (struct isc_table_type *isc,
                        wsio_addr_handle_t *cfg_handle);

PARAMETERS

isc Pointer to the driver's isc_table entry.

cfg_handle Pointer to contain the configuration handle upon completion.

DESCRIPTION

The wsio_map_cfg_handle() WSIO function is called by device drivers to obtain a handle to access configuration space. wsio_map_cfg_handle() must not be called in a non-blocking context.

RETURN VALUES

WSIO_OK Indicates a handle was successfully returned in cfg_handle.

WSIO_ERROR Indicates there was an error obtaining a handle.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

```c
wsio_addr_handle_t handle;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
  /* Error obtaining configuration space handle. Return an
   * error code */
  return(ERROR);
}
/* The configuration space handle is now in the handle
 * variable */
```

SEE ALSO

wsio_cfg_inXX(WSIO3), wsio_cfg_outXX(WSIO3),
wsio_unmap_cfg_handle(WSIO3)
NAME

wsio_map_dma_buffer(WSIO3) – Map an existing memory object for packet DMA.

SYNOPSIS

```
#include <wsio/wsio.h>

wsio_map_status_t wsio_map_dma_buffer (  
    struct isc_table_type * isc,  
    void * dma_handle,  
    wsio_map_context_t * context,  
    wsio_dma_buffer_hints_t hints,  
    wsio_range_type_t range_type,  
    wsio_dma_map_t * host_range,  
    wsio_dma_map_t * io_range);
```

PARAMETERS

- **isc**  
  Pointer to the driver's isc_table entry.
- **dma_handle**  
  DMA handle allocated using  
  `wsioAllocate dma_handle()`.
- **context**  
  Pointer to the context used for mapping.
- **hints**  
  Bitmask that provides mapping hints. The allowable hints are as follows:

<table>
<thead>
<tr>
<th>hints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_DMA_SAFE</td>
<td>Forces coherent transactions to be used even for full-cacheline transactions. In some implementations semi-coherent transactions are used to enhance performance when it is known that the entire cacheline will be modified. Affects inbound DMA only.</td>
</tr>
</tbody>
</table>
WSIO_DMA_LOCK

In some implementations this allows atomic access to memory for devices using bus-lock primitives.

WSIO_DMA_FLUSH_ON_USE

In some implementations this hint tells the hardware to flush resources associated with this buffer after they are used. This inhibits coalescing transactions into larger transfers for cases where it is not beneficial.

WSIO_DMA_IGN_ALIGNMENT

Indicates map_dma_buffer should not automatically set the IO_SAFE hint for partial cacheline buffers.

WSIO_DMA_CONTIGUOUS

Indicates map_dma_buffer must allocate a single contiguous IOVA. If wsio_map_dma_buffer is unable to do this, it will return WSIO_MAP_E_PARAMETER_ERROR. This hint implies WSIO_IO_IGN_ALIGNMENT.

WSIO_DMA_NO_SEQ

Inhibits hardware prefetching for outbound DMA in some implementations.

WSIO_DMA_INBOUND

Indicates the buffer will be used exclusively for inbound DMA.

WSIO_DMA_OUTBOUND

Indicates the buffer will be used exclusively for outbound DMA.

WSIO_DMA_NULL

Forces all hint values to zero. A hint value of zero tells the BN-CDIO to take hint values from the DMA object.
range_type  Indicates the type of host memory being mapped. It can be:

KERNELSPACE  Indicates host_range is a kernel virtual buffer.

PHYSICAL Indicates host_range is a physical buffer.

> 0 Indicates host_range is in user space, and this will be the space ID of the virtual address.

host_range  Pointer to an address/length structure that contains information about the host space to map. If the mapping was only partially completed, this will contain information about the remaining space to be mapped when the call completes.

io_range  Pointer to an address/length structure that will contain information about the I/O space that was mapped.

DESCRIPTION

The wsio_map_dma_buffer() WSIO function is called by a device driver to map an existing memory object for packet DMA. If continuous DMA is required wsio_allocate_shared_mem() should be used. Continuous DMA should be used for control structures, circular buffers, or any kind of buffer that needs to be accessed on a continuous basis by the I/O device. Packet DMA should be used when the mappings are temporary, or when pre-existing memory objects must be mapped for DMA.

If the entire host range cannot be mapped, the buffer may only be partially mapped. This will be indicated by a return value of WSIO_MAP_W_PARTIAL. In this case, wsio_map_dma_buffer() will need to be called again to map more of the host range. Using this method of calling wsio_map_dma_buffer() multiple times, the entire host range can be mapped into multiple I/O virtual ranges.

All mappings remain in effect until wsio_unmap_dma_buffer() or wsio_remap_dma_buffer() are called to remove or change them.

Callers are guaranteed buffers of at least 4K Bytes can be mapped into a contiguous range of I/O virtual address.
The `context` parameter is used to insure I/O resources are used efficiently. This structure should be initialized via a call to `wsio_init_map_context()`. The same context should be used for any group of mappings that will all be unmapped at the same time. If multiple buffers will not be mapped into one context, NULL can be passed in instead of a valid context.

Device drivers can set up a callback routine that will come into play if resources are not available at the time a mapping is attempted. If this callback is set up, `WSIO_MAP_W_CALLBACK` will be returned instead of a no resource error. When resources become available, the callback routine will be called to indicate this to the device driver. For more information on how to setup and use a callback, see the `wsio_set_dma_callback()` manpage.

`wsio_map_dma_buffer()` can be called in a non-blocking context.

**RETURN VALUES**

- **WSIO_MAP_OK**  
  Returned if the entire buffer has been mapped.

- **WSIO_MAP_W_PARTIAL**  
  Returned if only part of the buffer has been mapped.

- **WSIO_MAP_W_CALLBACK**  
  Returned if no resources are available and a callback function exists.

- **WSIO_MAP_E_NO_RESOURCES**  
  Returned if no resources are available and no callback function exists.

- **WSIO_MAP_E_RESOURCE_ERROR**  
  Returned if the request cannot and will never succeed.

- **WSIO_MAP_E_HIGH_ADDR**  
  Returned if the call failed because the device cannot reach the destination address.

- **WSIO_MAP_E_PARAMETER_ERROR**  
  Returned if an invalid parameter has caused failure of the call.

- **WSIO_MAP_E_UNKNOWN_ERROR**  
  Returned for hardware or other errors.
CONSTRANTS

EXAMPLE

```c
void *dma_handle;
wsio_map_context_t dma_con;
wsio_dma_map_t host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
wsio_init_map_context(&dma_con);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_map_dma_buffer(isc_entry,dma_handle,dma_con,
    WSIO_DMA_OUTBOUND,KERNELSPACE,&host_range,&io_range)
!= WSIO_MAP_OK) {
    /* Unable to perform the mapping so return an error */
    return(ERROR);
}

/* The host virtual buffer represented in the above
 * code by host_virtual_address’ is now mapped. Note that
 * this code does not handle the case where the buffer is
 * only partially mapped. In that case, a more
 * complete example would call wsio_map_dma_buffer() again
 * each time WSIO_MAP_PARTIAL was returned and save each
 * io_range that was returned.
 */
```

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_map_dma_buffer(WSIO3)

wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3),
wsio_unmap_dma_buffer(WSIO3),
NAME

wsio_map_port(WSIO3) – Obtain an I/O port handle.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_map_port (struct isc_table_type *isc,
                   int32_t port_addr,
                   size_t size,
                   wsio_addr_handle_t *port_handle);

PARAMETERS

isc          Pointer to the driver's isc_table entry.
port_addr    Address of the port obtained from
              wsio_get_iports().
size         Size of the port to be mapped.
port_handle  The port handle upon completion.

DESCRIPTION

The wsio_map_port() WSIO function will attempt to map an I/O port.
After a port has been mapped, it can be read from and written to using
the wsio_port_inXX() and wsio_port_outXX() accessor functions.
wsio_map_port() must not be called in a non-blocking context.

RETURN VALUES

WSIO_OK  Successful completion.
WSIO_ERROR  There was a parameter error.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

wsio_addr_handle_t port_handle;
wsio_iop_t iports_array[10];
/* An array with enough space for all ports needs to be
 * allocated */

if (wsio_get_ioports(isc_entry,10,iports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,iports_array[0].addr,
                 iports_array[0].size, &port_handle) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* The first I/O port will now be mapped and can be
 * accessed via 'port_handle'. */

SEE ALSO

wsio_get_ioports(WSIO3), wsio_port_inXX(WSIO3),
wsio_port_outXX(WSIO3), wsio_unmap_port(WSIO3)
NAME

`wsio_map_reg` (WSIO3) – Map device registers to host memory space.

SYNOPSIS

```
#include <wsio/wsio.h>

int wsio_map_reg (struct isc_table_type *isc,  
                 wsio_reg_info_t *reg_info);
```

PARAMETERS

- `isc` Pointer to the driver's isc_table entry.
- `reg_info` Pointer to information about the register to be mapped. This structure should be obtained from `wsio_get_all_registers()`. It will also be used after the mapping to access the register.

DESCRIPTION

The `wsio_map_reg()` WSIO function, given information about a register to be mapped, will attempt to map a register so that it is accessible via the `wsio_read_regXX()` and the `wsio_write_regXX()` functions. After the mapping has been performed, the `reg_info` variable can be used to access the register. Prior to the mapping, `reg_info` must be obtained from a call to `wsio_get_all_registers()`.

`wsio_map_reg()` must not be called in a non-blocking context.

RETURN VALUES

- `WSIO_OK` Successful completion.
- `WSIO_ERROR` Could not create the mapping.

CONSTRAINTS

Must not be called in an interrupt context.
EXAMPLE

wsio_reg_info_t *registers;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will
 * now be mapped. */

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_read_regXX(WSIO3),
wsio_write_regXX(WSIO3), wsio_unmap_reg(WSIO3),
WSIO Reference Pages

WSIO_ORDERED_INTERRUPTS(WSIO3)

NAME

WSIO_ORDERED_INTERRUPTS(WSIO3) – Macro to indicate whether interrupts are ordered with respect to DMA transactions.

SYNOPSIS

#include <wsio/wsio.h>

int WSIO_ORDERED_INTERRUPTS (struct isc_table_type * isc);

PARAMETERS

isc Pointer to the driver's isc_table entry.

DESCRIPTION

The WSIO_ORDERED_INTERRUPTS() macro reports to a device driver about whether interrupts are ordered with respect to DMA transactions. If true (1) is returned, nothing needs to be done. However, if false (0) is returned, interrupts are not ordered, and drivers must ensure DMA transactions have completed by reading a status register, or by calling wsio_io_sync().

WSIO_ORDERED_INTERRUPTS() can be called in a non-blocking context.

RETURN VALUES

Returns a one (1) if interrupts are ordered with respect to DMA transactions, and a zero (0) otherwise.

CONSTRAINTS

EXAMPLE

if (WSIO_ORDERED_INTERRUPTS(isc_entry)) {
    wsio_io_sync(isc_entry);
} else {
    /* No syncing necessary */
}
SEE ALSO

wsio_io_sync(WSIO3)
NAME

wsio_port_inXX(WSIO3) – Read XX bits from an I/O port.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_port_inXX (struct isc_table_type *isc,
                    wsio_addr_handle_t handle,
                    uint32_t addr,
                    uintXX_t *data);

PARAMETERS

isc Pointer to the driver's isc_table entry.
handle Handle obtained from wsio_map_port().
addr Offset into the port to read from.
data Contains the data read upon completion.

DESCRIPTION

The wsio_port_inXX() WSIO function is used to read XX from an I/O port. The port must have been mapped prior to the read by using the wsio_map_port() macro. No endian swapping is performed for port access, so if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap. The XX refers to the size of the transfer to perform and must be either 8, 16, 32, or 64.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t iports_array[10];
uint32_t data;

/* An array with enough space for all ports needs to be
allocated. We have just made it a local variable in this
example */

if (wsio_get_ioports(isc_entry,10,iports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,iports_array[0].addr,
                  iports_array[0].size, &port_handle
                  ) != WSIO_OK) {  
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now read 32-bits from the port */
wsio_port_in32(isc_entry,port_handle,0,&data);

/* 'data' will now contain whatever was at offset 0 of I/O
* port 0 */
```

SEE ALSO

wsio_get_ioports(WSIO3), wsio_map_port(WSIO3),
wsio_port_outXX(WSIO3), wsio_unmap_port(WSIO3)
NAME

wsio_port_outXX(WSIO3) – Write XX bits to an I/O port.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_port_outXX (struct isc_table_type *isc,
                     wsio_addr_handle_t handle,
                     uint32_t addr,
                     uintXX_t data);

PARAMETERS

isc Pointer to the driver's isc_table entry.
handle Handle obtained from wsio_map_port().
addr Offset into the port to read from.
data Data to be written.

DESCRIPTION

The wsio_port_outXX() WSIO function is used to write XX bits to an I/O port. The port must have been mapped prior to the write by using the wsio_map_port() macro. No endian swapping is performed for port access, so if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap. The XX refers to the size of the transfer to perform, and must be either 8, 16, 32, or 64.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t ioports_array[10];
uint32_t data;

/* An array with enough space for all ports needs to be
 * allocated. We have just made it a local variable in this
 * example */

if (wsio_get_ioports(isc_entry,10,ioports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,ioports_array[0].addr,
    ioports_array[0].size,
    &port_handle) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now write 32-bits to the port */
wsio_port_out32(isc_entry,port_handle,0,0x5a);

/* This example wrote the 32 bits 0x5a to I/O port 0 at
 * offset 0 */
```

SEE ALSO

wsio_get_ioports(WSIO3), wsio_map_port(WSIO3),
wsio_port_outXX(WSIO3), wsio_unmap_port(WSIO3)
NAME

wsio_probe_dev_info(WSIO4) – WSIO device probe information

SYNOPSIS

#include <wsio/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

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>Instance number of the nearest interface card ancestor.</td>
</tr>
<tr>
<td>target</td>
<td>Relative hardware address of first layer to be probed.</td>
</tr>
<tr>
<td>opt_1</td>
<td>Optional. Sometimes used for the hardware address of the second layer to be probed (e.g., LUN).</td>
</tr>
<tr>
<td>opt_2</td>
<td>Driver-discretionary element.</td>
</tr>
<tr>
<td>opt_3</td>
<td>Driver-discretionary element.</td>
</tr>
</tbody>
</table>
RETURN VALUES

CONSTRAINTS

EXAMPLE

A SCSI probe example might consist 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:

    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_query_supported_function(WSIO_DRV) – Return a pointer to a function supported by WSIO.

SYNOPSIS

#include <wsio/wsio.h>

void * wsio_query_supported_function (wsio_func_ptr_type_t func_id);

PARAMETERS

func_id Identifies what function pointer a driver is interested in.

func_id Function pointers returned

WSIO_GET_HANDLER_REG_FUNC
A WSIO function to register a driver's event handler.

WSIO_GET_HANDLER_UNREG_FUNC
A WSIO function to unregister a driver's event handler.

WSIO_GET_INSTALL_DRV_FUNC
A WSIO function to register a driver's function.

WSIO_GET_MASK_REG_FUNC
A WSIO function to register a driver's supported event mask.

DESCRIPTION

The wsio_query_supported_function() WSIO function queries WSIO to see if certain functions are supported. The purpose is that a driver can be installed on systems with different versions of wsio(11i or newer.) By doing this query, a driver can determine what functionality is supported and tailor its capabilities. If a function is supported, a pointer will be returned (otherwise a NULL); a driver can then call this function pointer. This mechanism will allow a driver to be able to link on multiple versions of wsio(11i or newer.)
RETURN VALUES

Returns a function pointer if supported, otherwise a NULL.

CONSTRAINTS

EXAMPLE

```c
static wsio_drv_info_t my_drv_info {
    ....
};
void my_event_handler()
{
    ....
}
int my_install(void)
{
    wsio_install_handler_func_ptr_t handler_reg_ptr;
    if (wsio_install_driver(&my_drv_info) {
        /* Query wsio to get a pointer to its registration function */
        handler_reg_ptr = wsio_query_supported_function
            (WSIO_GET_HANDLER_REG_FUNC);
        if (handler_reg_ptr) {
            /* Now register our event handler with wsio */
            ret = handler_reg_ptr(&my_drv_info, my_event_handler);
            ....
        }
    }
    ....
}
```

SEE ALSO

wsio_install_drv_event_handler(WSIO3),
wsio_install_drv_func(WSIO3),
wsio_reg_drv_capability_mask(WSIO5),
wsio_uninstall_drv_event_handler(WSIO3)
NAME

wsio_read_regXX(WSIO3) – Read XX bits from a mapped device register.

SYNOPSIS

#include <wsio/wsio.h>

void wsio_read_regXX (struct isc_table_type *isc,
                    wsio_reg_info_t *reg_info,
                    uint32_t offset,
                    uintXX_t *data);

PARAMETERS

isc            Pointer to the driver's isc_table entry.
reg_info       Pointer to information about the register to be read
                from. This register must have been successfully
                mapped by a call to wsio_map_reg().
offset         The offset into the register to read from.
data           Pointer to an appropriately sized data structure. This
                will contain the data upon completion of the read. The
                size should be selected based upon what size read is
                desired.

DESCRIPTION

The wsio_read_regXX() WSIO functions are called by device drivers to
read XX bits from a device register. The reg_info variable specifies
which register to read from, and offset indicates the correct location to
read from. The value XX refers to 8, 16, 32, or 64 and indicates the
amount of data to read. No endian swapping is performed for register
reads, so if the local bus and the host bus are of opposite endianness, the
driver must perform an endian swap.

RETURN VALUES

None.
CONSTRAINTS

EXAMPLE

```c
wsio_reg_info_t *registers;
uint32_t data;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}
/* All of the devices registers are now contained in
the 'registers' variable. They may be mapped as follows:
*/
if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}
/* The second device register (index 1 into the array) will
now be mapped. */
wsio_read_reg32(isc_entry, &registers[1],
    (uint32_t)0x10, &data);
/* 'data' will now contain 32 bytes obtained from offset
0x10 into the second register. */
```

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_map_reg(WSIO3),
wsio_unmap_reg(WSIO3), wsio_write_regXX(WSIO3)
NAME

`wsio_reg_drv_capability_mask(WSIO)` – Register a driver's capability mask.

SYNOPSIS

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

int wsio_reg_drv_capability_mask (struct isc_table_type *isc,
                                 wsio_event_mask_t event_mask);
```

PARAMETERS

- `isc` Pointer to the driver's `isc_table` entry.
- `event_mask` A mask representing the operations the driver will deal with in its event handler.

DESCRIPTION

The `wsio_reg_drv_capability_mask()` WSIO function is called by a device driver to register with WSIO for operations the driver is capable of handling. The mask is an OR of `wsio_event_t` and is for an instance of the driver. This function should be called in a driver's attach routine after `isc_claim()` is executed. The driver should register for the different events in one single call. A new mask will supercede the previous one.

If a platform does not support event handling, `WSIO_HA_NA` is returned and the driver should ignore the error.

See the Interrupt Migration chapter in the `Driver Development Guide` for relevant information.

RETURN VALUES

- `WSIO_OK` Returned on success.
- `WSIO_INVALID_COMBIN_EVENTS` Invalid combination of events.
- `WSIO_HA_NA` Platform does not support HA events.
WSIO Reference Pages
wsio_reg_drv_capability_mask(WSIO5)

WSIO_INVALID_EVENT
Invalid event.

WSIO_INVALID_ISC
Invalid isc.

WSIO_NO_DRV_HANDLER
A driver’s event handler has not been installed.

CONSTRAINTS

EXAMPLE

my_driver_attach() {
    int ret;
    wsio_event_mask_t my_mask = WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME | WSIO_EVENT_LBI_INTR_MIGR;
    isc_claim(isc, &my_drv_info);
    ret = wsio_reg_drv_capability_mask(isc, my_mask);
    if ( (ret!=WSIO_OK) && (ret!=WSIO_HA_NA)) {
        isc_unclaim(isc, &my_drv_info);
        //free resource as needed
    }
    // the rest of the attach chain
}

SEE ALSO

wsio_event_t(WSIO5), wsio_install_drv_event_handler(WSIO3), wsio_query_supported_function(WSIO_DRV)
NAME

wsio_register_addr_probe(WSIO3) – Register a driver probe function.

SYNOPSIS

#include<sys/wsio.h>

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 `wsio_register_addr_probe()` and the device driver would register its probe function via `wsio_register_dev_probe()`.

If the probe function is used as a standalone probe function then a NULL value is passed in as the second parameter. Most drivers need only register a single probe function using the WSIO service `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_t(WSIO4),
wsio_register_dev_probe(WSIO3),`
NAME

wsio_register_dev_probe(WSIO3) – Register a driver probe function

SYNOPSIS

#include<sys/wsio.h>

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 drv_path. Probe functions registered via the service wsio_register_dev_probe() should have the following calling syntax:

```
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_t structure of the driver that controls the device. desc is an ASCII string describing the device.

RETURN VALUES

0 Successful completion.
-1 Error.

EXAMPLE

```c
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_t(WSIO4),
wsio_register_addr_probe(WSIO3),
NAME

`wsio_remap_dma_buffer(WSIO3) – Map pre-allocated IOVAs to new host ranges.`

SYNOPSIS

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

wsio_map_status_t wsio_remap_dma_buffer (
    struct isc_table_type *isc,
    void *dma_handle,
    wsio_range_type_t range_type,
    wsio_dma_map_t *host_range,
    wsio_dma_map_t *io_range);
```

PARAMETERS

isc Pointer to the driver's `isc_table` entry.

dma_handle DMA handle allocated using `wsio_allocate_dma_handle()`.

range_type Indicates the type of host memory being mapped. It can be:

- KERNELSPACE Indicates `host_range` is a kernel virtual buffer.
- PHYSICAL Indicates `host_range` is a physical buffer.
- > 0 Indicates `host_range` is in user space, and this will be the space ID of the virtual address.

host_range Pointer to an address/length structure that contains information about the host space to map. If the mapping was only partially completed, this will contain information about the remaining space to be mapped when the call completes.

io_range Pointer to an address/length structure that will contain information about the I/O space that was mapped.
DESCRIPTION

The `wsio_remap_dma_buffer()` WSIO function is called by a device driver to map a new host memory address to existing I/O virtual addresses (IOVA). The IOVAs must have been previously allocated via a call to `wsio_map_dma_buffer()`, `wsio_remap_dma_buffer()`, or `wsio_fastmap_dma_buffer()`.

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 of equal sizes.

All mappings will remain in effect until `wsio_unmap_dma_buffer()` or `wsio_remap_dma_buffer()` are called to remove or change them.

A callback function is not necessary for `wsio_remap_dma_buffer()` because DMA resources were allocated when the initial mapping took place.

`wsio_remap_dma_buffer()` can be called in a non-blocking context.

RETURN VALUES

- **WSIO_MAP_OK**  
  Returned if the entire buffer has been mapped.

- **WSIO_MAP_E_HIGH_ADDR**  
  Returned if the call failed because the device cannot reach the destination address.

- **WSIO_MAP_E_PARAMETER_ERROR**  
  Returned if an invalid parameter has caused failure of the call.

- **WSIO_MAP_E_UNKNOWN_ERROR**  
  Returned for hardware or other errors.

CONSTRAINTS
EXAMPLE

void *dma_handle;
wsio_map_context_t dma_con;
wsio_dma_map_t host_range, new_host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
wsio_init_map_context(&dma_con);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_map_dma_buffer(isc_entry, dma_handle,
    dma_con, WSIO_DMA_OUTBOUND KERNELSPACE,
    &host_range, &io_range) != WSIO_MAP_OK) {
    return(INTERNAL_ERROR);
}

new_host_range.iov_base = new_host_virtual_address;
new_host_range.iov_len = dma_buffer_length;

if (wsio_remap_dma_buffer(isc_entry, dma_handle,
    KERNELSPACE, &new_host_range, &io_range
    ) != WSIO_MAP_OK) {
    return(INTERNAL_ERROR);
}

/* The host virtual buffer represented in the above
 * code by ’host_virtual_address’ was mapped. After that,
 * a second host virtual buffer represented by
 * ’new_host_virtual_address’ was mapped using the
 * same I/O virtual address as the initial mapping.
 * This example doesn’t take into account a
 * WSIO_MAP_W_PARTIAL return code being returned by the
 * call to wsio_map_dma_buffer().
 */
SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3), wsio_unmap_dma_buffer(WSIO3)
NAME

`wsio_set_description` (WSIO3) – Set the I/O tree node description for this driver.

SYNOPSIS

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

void wsio_set_description (struct isc_table_type *isc,
                           char *description);
```

PARAMETERS

- `isc` Pointer to the driver’s `isc_table` entry.
- `description` String containing the description.

DESCRIPTION

The `wsio_set_description()` WSIO function sets the I/O tree node description of a driver. This functionality is provided because at times, WSIO can not make appropriate decisions as to how to describe a driver at driver install time. This `description` is used by the `ioscan` command that is provided to system users, and therefore is essential to describing the system.

Many drivers may not need to call this function. To decide if it is necessary, a driver writer should make a decision based upon whether the `ioscan` output for that device driver is understandable.

RETURN VALUES

None.

CONSTRAINTS

EXAMPLE

```c
wsio_set_description (isc_entry,"My Driver Description");
```
WSIO Reference Pages

wsio_set_description(WSIO3)

SEE ALSO
NAME

\texttt{wsio\_set\_dma\_attributes}\texttt{(WSIO3)} – Associate DMA hints with a DMA handle.

SYNOPSIS

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

wsio_map_status_t wsio_set_dma_attributes(
    struct isc_table_type *isc,
    void *dma_handle,
    wsio_dma_attribute_t attribute,
    wsio_dma_attr_param_t param);
\end{verbatim}

PARAMETERS

- \textit{isc} Pointer to the driver's isc_table entry.
- \textit{dma_handle} DMA handle allocated using \texttt{wsio\_allocate\_dma\_handle()}.  
- \textit{attribute} Indicates which hint to set for the device associated with \textit{dma_handle}. The possible attributes are:

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_DMA_ATTR_ADDR_WIDTH</td>
<td>Bits of addressing supported by the device. This is used to determine whether a device can DMA directly to memory buffers. Default = 32</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_ALIGNMENT</td>
<td>Byte alignment of DMA buffer required for device. Default = H/W Dep.</td>
</tr>
</tbody>
</table>
WSIO_DMA_ATTR_ATM

ATM hint; used by hardware in some implementations.

0 = not ATM
1 = ATM48 (optimize for 48-byte transfers)
2 = ATM192 (optimize for 192-byte transfers)

Default = 0

WSIO_DMA_ATTR_CALLBACK

Specifies a function to call when resources become available.

Default = NULL

WSIO_DMA_ATTR_CALLBACK_ARG

Specifies an argument to the callback function.

Default = 0

WSIO_DMA_ATTR_FLUSH_ON_USE

Specifies the cacheline should be flushed from any intermediate buffers as soon as it is referenced. This inhibits any coalescing of data by bus bridges.

Default = 0

WSIO_DMA_ATTR_IGN_ALIGN

Specifies the mapping service should not handle cacheline fragments in a special way.

Default = 0
WSIO_DMA_ATTR_INTERLEAVE

IOVA allocation model

0 = DMA streams are normally interleaved (mass storage)
1 = DMA streams are normally not interleaved (networking)
3 = DMA buffers are static and accessed randomly (lowfat)

Default = 0

WSIO_DMA_ATTR_PREFETCH

Specifies how aggressively hardware should prefetch for outbound DMA.

0 = no prefetch
1 = moderate prefetch
2 = aggressive prefetch

Default = 1

WSIO_DMA_ATTR_SAFE

Specifies that the most conservative coherency model should be used for inbound DMA. Inhibits semi-coherent transactions such as WRITE_PURGE unless it is guaranteed that no data in processor caches will be lost.

1 = ON
2 = OFF

Default = 0

WSIO_DMA_ATTR_TXN_SIZE

Specifies the default transaction size used by the device. This is used by hardware to optimize conversion of transactions between busses.

Default = H/W Dep.
wsio_set_dma_attributes(WSIO3)

WSIO_DMA_ATTR_INBOUND
DMA buffers will be used exclusively for inbound DMA.
Default = 0

WSIO_DMA_ATTR_OUTBOUND
DMA buffers will be used exclusively for outbound DMA.
Default = 0

WSIO_DMA_ATTR_STABLE
Indicates that the buffer will not be modified by another entry while mapped for DMA.

\[
\begin{align*}
1 &= \text{true for data buffers} \\
0 &= \text{false for control structures}
\end{align*}
\]
Default = 0

\textit{param}
Driver defined parameter passed as the first parameter to \textit{isr}. Typically, \textit{isc} is passed as \textit{arg1}.

\section*{DESCRIPTION}

The \texttt{wsio\_set\_dma\_attributes()} WSIO function is used to associate DMA transaction hints and attributes with a specific DMA handle. These hints override any hints set via \texttt{wsio\_dma\_set\_device\_attributes()}, and are overridden by some hints passed in as parameters to \texttt{wsio\_map\_dma\_buffer().wsio\_set\_dma\_attributes()} can be called in a non-blocking context.

If WSIO_DMA_ATTR_INTERLEAVE is set to 1, a subsequent \texttt{wsio\_allocate\_shared\_mem()} or \texttt{wsio\_map\_dma\_buffer()} can only successfully request a buffer of a maximum of one page (4K) in length and this buffer cannot cross a page boundary. This is the limitation placed by the underlying platform. If a larger buffer is desirable, use the default value of 0. This larger buffer can be used for control structures rather than packet DMAs.

Do not call \texttt{wsio\_set\_dma\_attributes()} to set the WSIO_DMA_ATTR_INTERLEAVE to the default value of 0. The call will fail.
RETURN VALUES

WSIO_MAP_OK    Successful completion.
WSIO_MAP_E_PARAMETER_ERROR
    Returned if an invalid parameter has caused failure of the call.

CONSTRAINTS

EXAMPLE

if (wsio_set_dma_attributes(isc_entry, dma_handle,
                            WSIO_DMA_ATTR_INTERLEAVE, 1) != WSIO_MAP_OK) {
    /* There was a parameter error */
    return(ERROR);
} else {
    /* DMA streams are now not normally interleaved for all DMA associated with dma_handle */
    return(0);
}

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_dma_set_device_attributes(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_unmap_dma_buffer(WSIO3)
NAME

wsio_set_dma_callback(WSIO3) – Set the callback function and argument for DMA

SYNOPSIS

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

wsio_map_status_t wsio_set_dma_callback (struct isc_table_type *isc,
                                          void *dma_handle,
                                          void *func,
                                          void *arg);
```

PARAMETERS

- **isc**: Pointer to the driver’s isc_table entry.
- **dma_handle**: DMA handle allocated using `wsio_allocate_dma_handle`.
- **func**: Function pointer to be used as a callback.
- **arg**: Argument to be passed to the callback when it is called.

DESCRIPTION

The `wsio_set_dma_callback()` WSIO function is called by a device driver to setup the callback function for certain DMA transactions. If resources are not available when `wsio_map_dma_buffer()`, `wsio_fastmap_dma_buffer()`, or `wsio_allocate_shared_memory()` are called, and a callback function is setup, `WSIO_MAP_W_CALLBACK` is returned to the caller, and the callback function will be called when resources become available. This eliminates the need to continuously loop to attempt to obtain DMA resources.

RETURN VALUES

- **WSIO_MAP_OK**: Successful completion.
- **WSIO_MAP_E_PARAMETER_ERROR**: Returned if an invalid parameter has caused failure of the call.
CONRAINTS

EXAMPLE

if (wsio_set_dma_callback(isc_entry, dma_handle,
   callback_func, callback_arg)
   != WSIO_MAP_OK) {
  /* There was a parameter error */
  return(ERROR);
}

SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_map_dma_buffer(WSIO3),
wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3) wsio_unmap_dma_buffer(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

0 Successful completion.
<>0 Error.

CONSTRAINTS

SEE ALSO

wsio_install_driver(WSIO3)
NAME

wsio_uninstall_drv_event_handler(WSIO3) – Uninstall a driver’s event handler

SYNOPSIS

#include <wsio/wsio.h>

int wsio_uninstall_drv_event_handler (wsio_drv_t * drv_info,
                                 wsio_drv_event_handler_t drv_handler);

PARAMETERS

    drv_info     Pointer to the driver’s wsio_drv_info_t structure.
    drv_handler  Function pointer to the driver’s event handler

DESCRIPTION

The wsio_uninstall_drv_event_handler() WSIO function is called by a
driver to uninstall its event handler. The call is made if a driver needs
to clean up after an init failure, online deletion, or DLKM.

RETURN VALUES

    WSIO_OK            Returned on success.
    WSIO_DRV_NOT_FOUND Could not find the driver (driver has not registered
                        yet).
    WSIO_ERROR         Returned on failures.
    WSIO_INFO_NULL     Drv_info is NULL.
    WSIO_NO_DRV_HANDLER Invalid drive handler.

CONSTRAINTS
EXAMPLE

```
static wsio_drv_info_t my_drv_info {
    &my_info,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION,
}
my_cleanup() {
    int ret;
    ret = wsio_uninstall_drv_event_handler(
        &my_drv_info, my_handler);
    ...
    ret = wsio_uninstall_driver(&my_drv_info);
}
```

SEE ALSO

wsio_install_drv_event_handler(WSIO3)
NAME

`wsio_unmap_cfg_handle(WSIO3)` – Release a configuration space handle.

SYNOPSIS

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

int wsio_unmap_cfg (struct isc_table_type *isc,
                    wsio_addr_handle_t *cfg_handle);
```

PARAMETERS

- `isc` Pointer to the driver’s `isc_handle` entry.
- `cfg_handle` Pointer to the configuration handle.

DESCRIPTION

The `wsio_map_cfg_handle()` WSIO function is called by device drivers to release a configuration space handle.

RETURN VALUES

- **WSIO_OK** Indicates a handle was successfully returned in `cfg_handle`.
- **WSIO_ERROR** Indicates there was a parameter error and the handle has not been released.

CONSTRAINTS
EXAMPLE

wsio_addr_handle_t handle;

if (wsio_map_cfg_handle(isc_entry,&handle) != WSIO_OK) {
    /* Error obtaining configuration space handle. Return
        * an error code */
    return(ERROR);
}

if (wsio_unmap_cfg(isc_entry,&handle) != WSIO_OK) {
    /* Error releasing configuration space handle. Return an
        * error code */
    return(ERROR);
}

/* The configuration space handle is no longer valid */

SEE ALSO

wsio_map_cfg_handle(WSIO3), wsio_cfg_inXX(WSIO3),
wsio_cfg_outXX(WSIO3)
NAME

`wsio_unmap_dma_buffer` (WSIO3) – Remove a DMA packet mapping.

SYNOPSIS

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

wsio_map_status_t wsio_unmap_dma_buffer (
    struct isc_table_type *isc,
    void *dma_handle,
    wsio_dma_map_t *io_range);
```

PARAMETERS

- `isc` Pointer to the driver's `isc_table` entry.
- `dma_handle` DMA handle allocated using `wsio_allocate_dma_handle()`.
- `io_range` Pointer to an address/length structure that contains the information about the mapping to remove.

DESCRIPTION

The `wsio_unmap_dma_buffer()` WSIO function is called by a device driver to remove a packet DMA memory mapping, and to free all resources associated with such a mapping. `io_range` must have been obtained via a previous call to `wsio_map_dma_buffer()`, `wsio_remap_dma_buffer()`, or `wsio_fastmap_dma_buffer()`. `wsio_unmap_dma_buffer()` can be called in a non-blocking context.

RETURN VALUES

- `WSIO_MAP_OK` Successful completion.
- `WSIO_MAP_E_PARAMETER_ERROR` Returned if an invalid parameter has caused failure of the call. The buffer will not be unmapped.
EXAMPLE

```c
void *dma_handle;
wsio_map_context_t dma_con;
wsio_dma_map_t host_range, new_host_range, io_range;

dma_handle = wsio_allocate_dma_handle(isc_entry);
wsio_init_map_context(&dma_con);

host_range.iov_base = host_virtual_address;
host_range.iov_len = dma_buffer_length;

if (wsio_map_dma_buffer(isc_entry, dma_handle,
         dma_con, WSIO_DMA_OUTBOUND, KERNELSPACE, &host_range,
         &io_range) != WSIO_MAP_OK) {
    return(ERROR);
}

if (wsio_unmap_dma_buffer(isc_entry, dma_handle, &io_range
         ) != WSIO_MAP_OK) {
    /* There must have been a parameter error. */
    return(ERROR);
}

/* The host virtual buffer represented in the above
* code by 'host_virtual_address' was mapped and then
* immediately unmapped. This example doesn't take into
* account a WSIO_MAP_W_PARTIAL return code being
* returned by the call to wsio_map_dma_buffer(). */
```
SEE ALSO

wsio_allocate_dma_handle(WSIO3),
wsio_allocate_shared_mem(WSIO3), wsio_dma_pass_thru(WSIO3),
wsio_fastmap_dma_buffer(WSIO3), wsio_free_dma_handle(WSIO3),
wsio_free_shared_mem(WSIO3), wsio_flush_shared_mem(WSIO3),
wsio_init_map_context(WSIO3), wsio_iova_to_phys(WSIO3),
wsio_map_dma_buffer(WSIO3), wsio_remap_dma_buffer(WSIO3),
wsio_set_device_attributes(WSIO3),
wsio_set_dma_attributes(WSIO3)
NAME

wsio_unmap_port(WSIO3) – Unmap an I/O port.

SYNOPSIS

#include <wsio/wsio.h>

int wsio_unmap_port (struct isc_table_type *isc,
                     int32_t port_addr,
                     size_t size,
                     wsio_addr_handle_t port_handle);

PARAMETERS

isc Pointer to the driver's isc_table entry.
port_addr Address of the port obtained from
            wsio_get_iports().
size Size of the port.
port_handle Handle obtained from wsio_map_port().

DESCRIPTION

The wsio_unmap_port() WSIO function removes the mapping done by
wsio_map_port(). After this is called, the port should not be accessed by
the driver anymore.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR Parameter error.

CONSTRAINTS
EXAMPLE

```c
wsio_addr_handle_t port_handle;
wsio_iop_t iports_array[10];
/* An array with enough space for all ports needs to be
allocated */

if (wsio_get_ioports(isc_entry,10,iports_array) != WSIO_OK) {
    /* There was a problem obtaining the ports */
    return(ERROR);
}

if (wsio_map_port(isc_entry,iports_array[0].addr,
                   iports_array[0].size, &port_handle) != WSIO_OK) {
    /* There was an error mapping the port */
    return(ERROR);
}

/* Now unmmap the port */
if (wsio_unmap_port(isc_entry,iports_array[0].addr,
                    iports_array[0].size, port_handle) != WSIO_OK) {
    /* There was an error unmapping the port */
    return(ERROR);
}
```

SEE ALSO

wsio_get_ioports(WSIO3), wsio_map_port(WSIO3),
wsio_port_inXX(WSIO3), wsio_port_outXX(WSIO3)
NAME

\texttt{wsio\_unmap\_reg} (WSIO3) – Unmap a device register.

SYNOPSIS

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

int wsio_unmap_reg (struct isc_table_type *isc,
                    wsio_reg_info_t *reg_info);
\end{verbatim}

PARAMETERS

- \texttt{isc} Pointer to the driver's isc\_table entry.
- \texttt{reg\_info} Pointer to information about the register to be unmapped. This should be the same structure used to map the register.

DESCRIPTION

The \texttt{wsio\_unmap\_reg}() WSIO function removes the mapping done by \texttt{wsio\_map\_reg}().

RETURN VALUES

- \texttt{WSIO\_OK} Successful completion.
- \texttt{WSIO\_ERROR} Could not unmapped the register.

CONSTRAINTS
EXAMPLE

```c
wsio_reg_info_t *registers;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will now
 * be mapped. */

if (wsio_unmap_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register will now be unmapped */
```

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_map_reg(WSIO3),
wsio_read_regXX(WSIO3), wsio_write_regXX(WSIO3)
NAME

wsio_unregister_dev_probe(WSIO3) – Unregisters a driver probe function.

SYNOPSIS

int wsio_unregister_dev_probe (int type, char * name);

PARAMETERS

<table>
<thead>
<tr>
<th>type</th>
<th>Indicates what driver data the second parameter should be matched to. Valid values are:</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF_CLASS</td>
<td>The second argument, name is to be matched with the drv_path field of the wsio_drv_data_t structure.</td>
</tr>
<tr>
<td>DRV_NAME</td>
<td>The second argument, name is to be matched with the name field of the drv_info_t structure.</td>
</tr>
</tbody>
</table>

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_drv_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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successfully found and deleted the driver</td>
</tr>
<tr>
<td>-1</td>
<td>Not found</td>
</tr>
</tbody>
</table>

CONSTRAINTS

EXAMPLE

```c
int my_drv_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,"my_drv_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_t(WSIO4), wsio_drv_info(WSIO4),
wsio_register_dev_probe(WSIO3)
NAME

wsio_write_regXX(WSIO3) – Write XX bits to a mapped register.

SYNOPSIS

#include <wsio/wdio.h>

void wsio_write_regXX (struct isc_table_type *isc,
                      wsio_reg_info_t *reg_info,
                      uint32_t offset,
                      uintXX_t data);

PARAMETERS

isc                      Pointer to the driver's isc_table entry.
reg_info                Pointer to information about the register to be written to. This register must have been successfully mapped by a call to wsio_map_reg().
offset                  The offset into the register to write to.
data                    Appropriately sized piece of data to be written. The size should be selected based upon what size write is desired.

DESCRIPTION

The wsio_write_regXX() WSIO functions are called by device drivers to write XX bits to a device register. The reg_info variable specifies which register to write to, and offset indicates the correct location to write to. The value XX refers to 8, 16, 32, or 64 and indicates the amount of data to write. No endian swapping is performed for register writes, so if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap.

RETURN VALUES

None.
CONSTRANTS

EXAMPLE

```c
wsio_reg_info_t *registers;
uint32_t data = 0x5a;

registers = wsio_get_all_registers(isc_entry);
if (registers == NULL) {
    /* No registers exist. Return an error */
    return(ERROR);
}

/* All of the devices registers are now contained in the
 * 'registers' variable. They may be mapped as follows: */
if (wsio_map_reg(isc_entry,&registers[1]) != WSIO_OK) {
    return(ERROR);
}

/* The second device register (index 1 into the array) will now
 * be mapped. */
wsio_write_reg32(isc_entry, &registers[1],
    (uint32_t)0x10, data);
/* The 32 bytes consisting of 0x5a will now be written to
 * offset 0x10 into the second register. */
```

SEE ALSO

wsio_get_all_registers(WSIO3), wsio_map_reg(WSIO3),
wsio_read_regXX(WSIO3) wsio_unmap_reg(WSIO3)
Chapter 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.
Network Macros, Structures, and Functions
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 service 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) – 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_dlpis structure interface for support between HP-UX DLPI and device drivers.

**STRUCTURE**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hp_dlpis</strong></td>
<td>It must be initialized to all zeros by using bzero().</td>
</tr>
<tr>
<td><strong>mac_type</strong></td>
<td>A network media device type defined in /usr/conf/sio/lan_dlpikrn.h:</td>
</tr>
<tr>
<td></td>
<td>DEV_8023</td>
</tr>
<tr>
<td></td>
<td>DEV_8025</td>
</tr>
<tr>
<td></td>
<td>DEV_ATM</td>
</tr>
<tr>
<td></td>
<td>DEV_ETHER</td>
</tr>
<tr>
<td></td>
<td>DEV_FC</td>
</tr>
<tr>
<td></td>
<td>DEV_FDDI</td>
</tr>
<tr>
<td><strong>llc_flags</strong></td>
<td>Link Level Control (LLC) encapsulation method. The flag values defined in</td>
</tr>
<tr>
<td></td>
<td>/usr/conf/sio/lan_dlpikrn.h:</td>
</tr>
<tr>
<td></td>
<td>ETHERTYPE</td>
</tr>
<tr>
<td></td>
<td>IEEE</td>
</tr>
<tr>
<td></td>
<td>NOVELL</td>
</tr>
<tr>
<td></td>
<td>SNAP</td>
</tr>
<tr>
<td><strong>mjr_num</strong></td>
<td>Major number of the device file associated with the device. The major number</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>nm_id</strong></td>
<td>Network management ID. <strong>nm_id</strong> should be initialized via a call to the</td>
</tr>
<tr>
<td></td>
<td>get_nmid() routine.</td>
</tr>
<tr>
<td><strong>instance_num</strong></td>
<td>Device instance number. This number is in the value returned by calling</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: <strong>The wsio_isc_to_instance()</strong> routine cannot be called from a</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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>
<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_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
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:

DRV_MBLK
This flag must be set since the third
party network driver is purely based
on STREAMS model.

DRV_MP
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.

DRV_IP_MULTICAST
This flag must be set if driver
supports IP multicast feature.

DRV_LANC_PROMISC_SUPPORT
This flag must be set if driver
supports promiscuous listening.

DRV_NO_FAST_PATH
This flag must be set if driver does
not support fast path as described in
“Transmission of Message Blocks”.

DRV_CKO
This flag must be set if driver
supports TCP or UDP checksum
calculations in hardware.

**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_ift structure. It is initialized in hw_ift_attach().

next  Pointer to the next hw_ift 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) – 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);

PARAMETERS

hw_ift_ptr Pointer to the associated hw_ift structure.

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()

CONSTRAINTS

SEE ALSO

driver_if_init(WSIO_DRV), hw_ift(NET4)
NAME

HWIFT_LOCK(NET3), HWIFT_UNLOCK(NET3) – Acquire/release hwift_lock spinlock

SYNOPSIS

#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 services 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 hwift_lock in cases where the dlpi_ioctl() request is not able to complete immediately.

RETURN VALUES

None.
HWIFT_LOCK(NET3), HWIFT_UNLOCK(NET3)

CONSTRAINTS

SEE ALSO

hw_ift (NET4), hw_ift_attach (NET3)
NAME

kget_log_instance(NET3) – 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

n A unique number for use as a log instance value.

CONSTRAINTS

SEE ALSO

klogg_write(NET3)
NAME

KLOG_CK(NET3) – 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 service is a macro that allows the calling process to find out whether logging is enabled for the current subsystem.

RETURN VALUES

- 0: Logging is disabled.
- 1: Logging is enabled.

CONSTRAINTS
SEE ALSO

klogg_write(NET3)
NAME
klogg_write(NET3) – Send log messages to the kernel trace and log facility

SYNOPSIS

```c
#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.
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.

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)_ – Check whether tracing is enabled; verify tracing activation on a per-interface-device basis.

SYNOPSIS

```c
#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 service 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

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,
    (&lan2_ift_ptr->lancift.is_if)->if_unit))
{
    ktrc_write(...);
}
```

SEE ALSO

ktrc_write(NET3)
NAME

ktrc_write(NET3) – Send trace messages to kernel trace and log facility

SYNOPSIS

#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) – Set up global information for the filter and formatting functions

SYNOPSIS

```
#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.
CONSTRAINTS

SEE ALSO

set_up_ether(NET3), set_up_ip(NET3), set_up_link(NET3)
NAME

set_up_ether(NET3) – 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_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) – 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
SEEE ALSO

set_up_8022(NET3), set_up_ether(NET3), set_up_link(NET3)
**NAME**

*set_up_link*(NET3) – 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) – Routine to format a single trace or log message from subsystem N.

SYNOPSIS

```c
#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 map_to_names_bit flag is set.

binary-msg-ptr

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.
**options-ptr**  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 `nettigen.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

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) – Routine to process options for subsystem N

SYNOPSIS

```c
#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.

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_optParms_type structure is defined in <fmt.h> as:

typedef struct
{
  int         *status_ptr;
  FILE        *subsys_strm;
  FILE        *errno_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_optParms_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.

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

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)` – 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) – 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

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

tl_header_format1(NET3)
NAME

tl_format_fprintf(NET3) – 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 tl_format_fprintf() network function converts, formats, and prints its arguments under control of the format. This routine behaves like printf() but must be used instead to give netfmt() control over the formatted buffer.

RETURN 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 the error_fd parameter of the subsys_N_format() routine.

FMTERR_FORMAT_FPRINTF An error occurred in writing to the output stream.

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
Network Device Driver Reference Pages

\texttt{tl\_format\_fprintf(NET3)}

\textbf{SEE ALSO}

\texttt{subsys\_N\_format(NET\_DRV), tl\_format\_write(NET3),}
\texttt{tl\_get\_parms(NET3), tl\_raw\_format(NET3)}
NAME

tl_format_write(NET3) – Write a buffer to standard output

SYNOPSIS

#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().

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.
reserved

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, and the value is always 1.

output_files[]
An array of structures consisting of a file descriptor and 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 refers to the file receiving the formatter output.

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.

status_ptr
Contains the error value if the return value is -1.

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 sprintf() C library function.

RETURN 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)` – Obtain a line from a filter command file

SYNOPSIS

```c
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

2 Successful completion.
0 End of file.
<0 Error.

CONSTRAINTS

SEE ALSO

subsys_N_get_options(NET_DRV)
NAME

```
tl_get_parms(NET3) – 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:

```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**: 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

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

subsys_N_format(NET_DRV)
NAME

tl_header_format1(NET3) – Routine to format a single trace or log header

SYNOPSIS

#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 \texttt{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: \texttt{output\_file[0].fd}.

time_buffer  A string depicting the formatted time stamp from the trace/log header.

time_buffer_length  The length of \texttt{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 \texttt{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:

\begin{verbatim}
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.
\end{verbatim}
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

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) – 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

<>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) – 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; output_file[0].fd refers to the file receiving the formatter output.

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.

status_ptr  The error value if the routine returns -1.

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

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) – Return a text interpretation for a trace kind value

SYNOPSIS

#include <sys/subsys_id.h>

char * tl_trace_kind (unsigned int kind);

PARAMETERS

kind The numeric trace kind of the message. The keywords are defined in <sys/subsys_id.h>.

kind Value and Keyword Description

0x80000000 HDR_IN_BIT Inbound Protocol Header.
0x40000000 HDR_OUT_BIT Outbound Protocol Header.
0x20000000 PDU_IN_BIT Inbound Protocol Data Unit (including header and data).
0x10000000 PDU_OUT_BIT Outbound Protocol Data Unit (including header and data).
0x08000000 PROCEDURE_TRACE_BIT Procedure entry and exit.
0x04000000 STATE_TRACE_BIT Protocol or connection states.
0x02000000 ERROR_TRACE_BIT Invalid events or condition.
0x01000000 LOGGING_TRACE_BIT Special kind of trace that contains a log message.
0x00800000 LOOP_BACK_BIT Packets whose source and destination system are the same.
0x00400000 PTOP_BIT Packets whose transmission is point to point.
tl_trace_kind(NET3)

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

<> 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)`
Chapter 6  PCI Reference Pages
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) – Associate an initialization routine with a driver

SYNOPSIS

#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.
NAME

PCI_ATTACH_DEV_INIT_ERROR(PCI3) – 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) – 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_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) – Get field-replaceable-unit (FRU) information for the device associated with an ISC

SYNOPSIS

#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 Reference Pages
pci_get_fru_info_isc(PCl3)

RETURN 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) – Obtain a system-defined handle for manipulating a range of PCI I/O ports

SYNOPSIS

#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

0 Failure. A handle could not be returned by the system.
1 Success. The value pointed to by phndl is a valid handle.
CONSTRANTS

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) – 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 `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

`pci_write_cfg_uintN_isc(PCI3)`
NAME

pci_read_port_uintN_isc(PCI3) – 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
cpci_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

#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.hnd1;
} 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) – 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

- 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) – Write unsigned integer to a PCI configuration register

SYNOPSIS

#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.
RETURN VALUES

The pci_write_cfg_uintN_isc() routines do not return values.

CONSTRAINTS

SEE ALSO

pci_read_cfg_uintN_isc(PCI3)
NAME

pci_write_port_uintN_isc(PCI3) – 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) – 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.

data A pointer to an 8-, 16-, or 32-bit location where the routine should place the resultant data.

DESCRIPTION

The READ_REG_UINTn_ISC() PCI services are macros that read and byte-swap data located at addr from a little-endian bus and place it in 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 READ_REG_UINTn_ISC() if you define the flag PCI_LITTLE_ENDIAN_ONLY prior to including the pci.h header file. This causes READ_REG_UINTn_ISC() to perform a simple byte swap instead of calling a function that tests byte ordering.

RETURN VALUES

The 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) – 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isc</code></td>
<td>Pointer to an ISC table associated with the device.</td>
</tr>
<tr>
<td><code>addr</code></td>
<td>A pointer to the output address. It must be one of the following:</td>
</tr>
<tr>
<td></td>
<td>- A virtual address mapped with <code>map_mem_to_host()</code>.</td>
</tr>
<tr>
<td></td>
<td>- A mapped offset in the automatically mapped first-base-address register range contained in <code>isc-&gt;if_reg_ptr</code>.</td>
</tr>
</tbody>
</table>
— 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} \hspace{1cm} The 8-, 16-, or 32-bit data to be written.

**DESCRIPTION**

The \texttt{WRITE_REG_UINTn_ISC()} PCI services 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 \texttt{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 \texttt{WRITE_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{WRITE_REG_UINTn_ISC()} to perform a simple byte swap instead of calling a function that tests byte ordering.

**RETURN VALUES**

The \texttt{WRITE_REG_UINTn_ISC()} routines do not return values.

**CONSTRAINTS**
EXAMPLES

```c
#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 \texttt{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 \texttt{wsio map()} with flags \texttt{IO NO SEQ} and \texttt{IO SAFE} set, regardless of the type of transactions the PCI master uses.

3. The mapping is done with \texttt{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 IO TLB was mapped using either \texttt{wsio fastmap()} or \texttt{wsio map} without \texttt{IO NO SEQ} and \texttt{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 its 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 its 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.
Chapter 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 provide a supported pass-through mechanism.
SCSI Functions
NAME

dd_close(SCSI_DRV) – SCSI driver entry point to handle device close.

SYNOPSIS

void dd_close (dev_t dev);

PARAMETERS

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

DESCRIPTION

The dd_close() SCSI function is provided by the driver writer. It can
have any unique name. 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

None.

CONSTRAINTS

SEE ALSO

scsi_lun_close(SCSI3), scsi_ddsw(SCSI4)
NAME

`dd_done(SCSI_DRV)` – SCSI driver entry point to handle post-I/O processing

SYNOPSIS

```c
int dd_done (struct buf *bp);
```

PARAMETERS

`bp` Pointer to a buf structure.

DESCRIPTION

The `dd_done()` SCSI function is provided by the driver writer. It can have any unique name. 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 value is not used by SCSI services.

SEE ALSO

`biodone(KER2), scsi_action(SCSI3), scsi_ddsw(SCSI4)`
NAME

dd_ioctl(SCSI_DRV) – SCSI driver entry point to handle device I/O controls.

SYNOPSIS

int dd_ioctl (dev_t dev, int cmd, caddr_t data, int flags);

PARAMETERS

dev Device number

cmd Command word

data Pointer to the command parameters

flags File access flags.

DESCRIPTION

The dd_ioctl() SCSI function is provided by the driver writer. It can have any unique name. 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

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 driver entry point to allow/disallow ioctl commands sent through the pass-through driver

SYNOPSIS

int dd_ioctl_okay (dev_t dev, int cmd, caddr_t data, int flags);

PARAMETERS

dev Device number
cmd Command word
data Pointer to command parameter
flags File access flags

DESCRIPTION

The dd_ioctl_okay() SCSI function is provided by the driver writer. It can have any unique name. 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

PT_OKAY Successful completion.
0 Error.

CONSTRAINTS

SEE ALSO

scsi_ioctl(SCSI3)
NAME

`dd_open(SCSI_DRV)` – SCSI driver entry point to handle device open.

SYNOPSIS

```c
dd_open (dev_t dev, int oflags);
```

PARAMETERS

- `dev` Device number of the device to be opened
- `oflags` Flags passed in the open call

DESCRIPTION

The `dd_open()` SCSI function is provided by the driver writer. It can have any unique name. 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

- `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 driver entry point to handle post-pass-through I/O processing.

SYNOPSIS

```c
int dd_pass_thru_done (struct buf *bp);
```

PARAMETERS

- `bp` Pointer to a `buf` structure

DESCRIPTION

The `dd_pass_thru_done()` SCSI function is provided by the driver writer. It can have any unique name. 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 driver entry point to control pass-through I/O requests.

SYNOPSIS

```
dd_pass_thru_okay (dev_t dev, struct sctl_io * sctl_io);
```

PARAMETERS

- `dev` Device number
- `sctl_io` Struct containing ioctl information

DESCRIPTION

The `dd_pass_thru_okay()` SCSI function is provided by the driver writer. It can have any unique name. 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

- `PT_OKAY` Successful completion.
- `0` Error.

CONSTRAINTS
NAME

`dd_read(SCSI_DRV)` – SCSI driver entry point to handle device read operations.

SYNOPSIS

```c
int dd_read (dev_t dev, struct uio * uiop);
```

PARAMETERS

- `dev`  
  Device number

- `uiop`  
  Pointer to a `uio` structure.

DESCRIPTION

The `dd_read()` SCSI function is provided by the driver writer. It can have any unique name. 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

- `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 driver entry point to start an I/O request.

SYNOPSIS

struct buf * dd_start(struct scsi_lun *lp, struct scb *scb);

PARAMETERS

lp Pointer to the scsi_lun structure.
scbp Pointer to the SCSI control block.

DESCRIPTION

The dd_start() SCSI function is provided by the driver writer. It can have any unique name. 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

<>NULL Successful completion.
NULL Error.

CONSTRAINTS
NAME

`dd_strategy(SCSI_DRV)` – SCSI driver entry point to handle buf requests.

SYNOPSIS

```c
int dd_strategy (struct buf *bp struct scsi_lun *lp);
```

PARAMETERS

- `*bp` Pointer to a `buf` structure
- `*lp` Pointer to a `scsi_lun` structure.

DESCRIPTION

The `dd_strategy()` SCSI function is provided by the driver writer. It can have any unique name. 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

- `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 driver entry point to handle device write operations.

SYNOPSIS

```
int dd_write (dev_t dev, struct uio * uiop);
```

PARAMETERS

- **dev**: Device number
- **uiop**: Pointer to a uio structure.

DESCRIPTION

The _dd_write_() SCSI function is provided by the driver writer. It can have any unique name. 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

- **0**: Successful completion.
- **<>0**: Error. The value is expected to be an errno value.

SEE ALSO

physio(KER2), scsi_write(SCSI3)
NAME

\texttt{driver\_if\_abort}(SCSI\_DRV2) – Interface driver specific SCSI abort function.

SYNOPSIS

\texttt{int driver\_if\_abort (dev\_t dev);} 

PARAMETERS

\begin{itemize}
  \item \texttt{dev} \hspace{1cm} The device number
\end{itemize}

DESCRIPTION

The SCSI subsystem allows, but does not require, the interface driver to specify an abort function. It can have any unique name. Pass the name SCSI services by specifying it in the \texttt{if\_abort} field of the \texttt{scsi\_ifsw} structure. Commonly, the driver is replaced by your driver’s name.

It is intended to serve as a way for the SCSI subsystem to direct the interface driver to send a SCSI ABORT message to the indicated logical unit. The SCSI subsystem makes this call only in response to an \texttt{SIOC\textunderscore ABORT} ioctl request.

See the \textit{HP-UX Driver Development Guide} for details.

RETURN VALUES

\begin{itemize}
  \item 0 \hspace{1cm} Success.
  \item \textless{}0 \hspace{1cm} Error.
\end{itemize}

CONSTRAINTS

SEE ALSO

\texttt{scsi\_ifsw(SCSI3)}
NAME

driver_if_bdr(SCSI_DRV2) – Interface driver specific SCSI Bus Device Reset function.

SYNOPSIS

int driver_if_bdr (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

The SCSI subsystem allows, but does not require, the interface driver to specify a Bus Device Reset (BDR) function. It can have any unique name. Pass the name SCSI services by specifying it in the if_bdr field of the scsi_ifsw structure. Commonly, the driver is replaced by your driver's name.

It is intended to serve as a way for the SCSI subsystem to direct the interface driver to send a SCSI BDR message to the indicated target. The SCSI subsystem makes this call only in response to an SIOC_RESET_DEV ioctl request.

See the HP-UX Driver Development Guide for details.

RETURN VALUES

0 Success.
<>0 Error.

CONSTRAINTS

SEE ALSO

scsi_ifsw(SCSI3)
NAME

driver_if_close (SCSI_DRV2) – Interface driver specific logical unit close processing

SYNOPSIS

int driver_if_close (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

The SCSI subsystem allows, but does not require, the interface driver to specify a logical unit close function. It can have any unique name. You pass the name SCSI services by specifying it in the if_close field of the scsi_ifsw structure. Commonly, the driver is replaced by your driver’s name.

On all logical closes, the SCSI subsystem checks the if_close field of the scsi_ifsw structure for the SCSI bus. If the if_close field is not NULL, the SCSI subsystem calls it with the device number of the device being opened as its sole argument.

See the HP-UX Driver Development Guide for details.

RETURN VALUES

0 Success.
<0 Error.

CONSTRAINTS

It is never called under interrupt context. It is allowed to sleep. The SCSI subsystem provides protection that blocks all other opens and closes to the same logical unit until it returns.

SEE ALSO

scsi_ifsw(SCSI3)
NAME

driver_if_open(SCSI3) – Interface driver specific logical unit open processing

SYNOPSIS

int driver_if_open (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

The SCSI subsystem allows, but does not require, the interface driver to specify a logical unit open function. It can have any unique name. You pass the name SCSI services by specifying it in the if_open field of the scsi_ifsw structure. Commonly, driver is replaced by your driver’s name.

On all logical opens, the SCSI subsystem checks the if_open field of the scsi_ifsw structure for the SCSI bus. If the if_open field is not NULL, the SCSI subsystem calls it with the device number of the device being opened as its sole argument.

See the HP-UX Driver Development Guide for details.

RETURN VALUES

0 Success.
<>0 Error.

CONSTRAINTS

It is never called under interrupt context. It is allowed to sleep. The SCSI subsystem provides protection that blocks all other opens and closes to the same logical unit until it returns.

SEE ALSO

scsi_ifsw(SCSI3)
NAME

\texttt{driver\_if\_reset\_bus} (SCSI\_DRV2) – Interface driver specific SCSI bus reset function.

SYNOPSIS

\begin{verbatim}
void driver_if_reset_bus (dev_t dev);
\end{verbatim}

PARAMETERS

\begin{verbatim}
dev The device number
\end{verbatim}

DESCRIPTION

The SCSI subsystem allows, but does not require, the interface driver to specify a bus reset function. It can have any unique name. You pass the name SCSI services by specifying it in the \texttt{if\_reset\_bus} field of the \texttt{scsi\_ifsw} structure. Commonly, the driver is replaced by your driver's name.

When the SCSI subsystem wants to reset a bus, it checks the \texttt{if\_reset\_bus} field of the \texttt{scsi\_ifsw} structure for the bus. If the \texttt{if\_reset\_bus} is not NULL, it is called with a device number identifying the bus as its sole argument. When \texttt{if\_reset\_bus} returns, the SCSI bus should have been reset.

I/Os that are disconnected and the I/Os that are connected with the bus (if any) at the time of the reset should be returned to the SCSI subsystem with the appropriate status field set to \texttt{SCTL\_INCOMPLETE}. That is, if it was the Request Sense resulting from a check condition that was terminated by the reset, then \texttt{scb->sense\_action} should be set to \texttt{SCTL\_INCOMPLETE}. Otherwise, \texttt{scb->cdb\_status} should be set to \texttt{SCTL\_INCOMPLETE}. “struct scb” is described under data structures later in this section.

The SCSI subsystem makes this call only in response to \texttt{SIOC\_RESET\_BUS} ioctl request.

See \textit{HP-UX Driver Development Guide} for details.
SCSI Reference Pages

driver_if_reset_bus(SCSI_DRV2)

RETURN VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>&lt;&gt;0</td>
<td>Error</td>
</tr>
</tbody>
</table>

CONSTRAINTS

SEE ALSO

scki_ifsw(SCSI3)
NAME

driver_if_start(SCSI_DRV2) – Interface driver specific start function

SYNOPSIS

void driver_if_start (struct isc_table_type *isc);

PARAMETERS

isc A pointer to structure isc_table_type

DESCRIPTION

The SCSI subsystem requires the interface driver to specify a start function. Its purpose is to ensure the SCSI subsystem does not hang by providing a way to inform the interface driver that it has work to do.

It can have any unique name. You pass the name SCSI services by specifying it in the if_start field of the scsi_ifsw structure. Commonly, the driver is replaced by your driver's name.

Whenever the SCSI subsystem enqueues an I/O on the select queue of a dormant bus, the function pointed by the if_start field of the scsi_ifsw structure for the bus is always called with a pointer to the isc_table_type structure as its sole argument.

The SCSI subsystem may call if_start at any time, i.e., when the bus is dormant or not, and in a process’ context or under interrupt. In all cases, the interface driver must continue to execute I/Os that are on the select queue until the bus becomes dormant.

A bus is considered dormant if it has no active I/Os. An I/O is considered to be active from the time it is enqueued on the select queue until scsi_cbfn is called for the I/O.

See the HP-UX Driver Development Guide for details.

RETURN VALUES

None
CONSTRAINTS

if_start is not permitted to sleep under any circumstances.

SEE ALSO

scsi_ifsw(SCSI3)
NAME

m_bus_id(SCSI3) – Returns SCSI Bus ID.

SYNOPSIS

#include<sys/scsi_ctl.h>
m_bus_id (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

m_bus_id() macro evaluates to the bus ID of the SCSI bus corresponding to dev.

RETURN VALUES

m_bus_id() does not return any values. It is a macro.

CONSTRAINTS
EXAMPLES

#include <sys/scsi_ctl.h>

static int
mydriver_lun_open(dev_t dev)
{
    struct isc_table_type * isc;
    int bus_id, tgt_id, lun_id;
    ....

    /* Get the SCSI bus ID */
    bus_id = m_bus_id(dev);

    /* Get the SCSI target ID */
    tgt_id = m_tgt_id(dev);

    /* Get the SCSI LUN ID */
    lun_id = m_lun_id(dev);

    ....
}

SEE ALSO
NAME

m_lun_id(SCSI3) – Returns SCSI LUN ID.

SYNOPSIS

#include<sys/scsi_ctl.h>
m_lun_id (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

m_lun_id() macro evaluates to the LUN ID of the SCSI LUN corresponding to dev.

RETURN VALUES

m_lun_id() does not return any values. It is a macro.

CONSTRAINTS
EXAMPLES

#include <sys/scsi_ctl.h>

static int
mydriver_lun_open(dev_t dev)
{
    struct isc_table_type * isc;
    int bus_id, tgt_id, lun_id;
    ....

    /* Get the SCSI bus ID */
    bus_id = m_bus_id(dev);

    /* Get the SCSI target ID */
    tgt_id = m_tgt_id(dev);

    /* Get the SCSI LUN ID */
    lun_id = m_lun_id(dev);

    ....
}

SEE ALSO
NAME

m_tgt_id(SCSI3) – Returns SCSI target ID.

SYNOPSIS

#include<sys/scsi_ctl.h>
m_tgt_id (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

m_tgt_id() macro evaluates to the target ID of the SCSI target corresponding to dev.

RETURN VALUES

m_tgt_id() does not return any values. It is a macro.

CONSTRAINTS
EXAMPLES

#include <sys/scsi_ctl.h>

static int
mydriver_lun_open(dev_t dev)
{
    struct isc_table_type * isc;
    int bus_id, tgt_id, lun_id;
    ....

    /* Get the SCSI bus ID */
    bus_id = m_bus_id(dev);

    /* Get the SCSI target ID */
    tgt_id = m_tgt_id(dev);

    /* Get the SCSI LUN ID */
    lun_id = m_lun_id(dev);

    ....
}

SEE ALSO
NAME

m_scsi_bus(SCSI3) – Returns scsi_bus pointer

SYNOPSIS

#include <sys/scsi_ctl.h>

struct scsi_bus * m_scsi_bus (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

m_scsi_bus() function returns the scsi_bus pointer corresponding to dev.

RETURN VALUES

NULL Error.

<>0 Pointer to the SCSI bus structure associated with dev.

CONSTRAINTS
EXAMPLES

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

static int
mydriver_if_abort(struct buf *bp)
{
    dev_t dev;
    struct scsi_bus *busp;
    struct scsi_tgt *tgtp;
    struct scsi_lun *lunp;
    ....

    /* Get the device number */
    dev = bp->b_dev;

    /* Get the pointer to scsi_bus structure */
    busp = m_scsi_bus(dev);

    /* Get the pointer to scsi_tgt structure */
    tgtp = m_scsi_tgt(dev);

    /* Get the pointer to scsi_lun structure */
    lunp = m_scsi_lun(dev);

    if(busp == NULL) {
        msg_printf("mydriver - a NULL scsi_bus
pointer\n");
        return (ENXIO);
    }

    ....
}
```

SEE ALSO
NAME

\texttt{m\_scsi\_isc(SCSI3)} – Returns \texttt{isc\_table\_type} pointer

SYNOPSIS

\begin{verbatim}
#include<sys/scsi_ctl.h>
struct isc_table_type * m_scsi_isc( dev_t dev);
\end{verbatim}

PARAMETERS

dev The device number

DESCRIPTION

\texttt{m\_scsi\_isc()} function returns the \texttt{isc\_table\_type} pointer corresponding to \texttt{dev}.

RETURN VALUES

\begin{itemize}
\item \texttt{NULL} Error.
\item \texttt{<>0} Pointer to the ISC structure associated with \texttt{dev}.
\end{itemize}

CONSTRAINTS
EXAMPLES

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

static int
mydriver_lun_open(dev_t dev)
{
    struct isc_table_type *isc;
    ..... 

    /* Get the isc pointer */
    isc = m_scsi_isc(dev);

    if(isc == NULL) {
        msg_printf("mydriver - a NULL isc returned\n");
        return (ENXIO);
    }

    ...
}
```

SEE ALSO
NAME

m_scsi_lun(SCSI3) – Returns scsi_lun pointer

SYNOPSIS

#include <sys/scsi_ctl.h>

struct scsi_lun * m_scsi_lun (dev_t dev);

PARAMETERS

dev The device number

DESCRIPTION

m_scsi_lun() function returns the scsi_lun pointer corresponding to dev.

RETURN VALUES

NULL Error.

<>0 Pointer to the SCSI LUN structure associated with dev.

CONSTRAINTS
EXAMPLES

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

static int
mydriver_if_abort(struct buf *bp)
{
    dev_t dev;
    struct scsi_bus *busp;
    struct scsi_tgt *tgtp;
    struct scsi_lun *lunp;
    ....

    /* Get the device number */
    dev = bp->b_dev;

    /* Get the pointer to scsi_bus structure */
    busp = m_scsi_bus(dev);

    /* Get the pointer to scsi_tgt structure */
    tgtp = m_scsi_tgt(dev);

    /* Get the pointer to scsi_lun structure */
    lunp = m_scsi_lun(dev);

    if (busp == NULL) {
        msg_printf("mydriver - a NULL scsi_bus pointer\n");
        return (ENXIO);
    }

    ....
}
```

SEE ALSO
**NAME**

*m_scsi_tgt*(SCSI3) – Returns scsi_tgt pointer

**SYNOPSIS**

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

struct scsi_lun * m_scsi_tgt (dev_t dev);
```

**PARAMETERS**

- `dev` The device number

**DESCRIPTION**

The `m_scsi_tgt()` function returns the scsi_tgt pointer corresponding to `dev`.

**RETURN VALUES**

- `NULL` Error.
- `<0` Pointer to the SCSI target structure associated with `dev`.

**CONSTRAINTS**
EXAMPLES

#include <sys/scsi_ctl.h>

static int
mydriver_if_abort(struct buf *bp)
{
    dev_t dev;
    struct scsi_bus *busp;
    struct scsi_tgt *tgtp;
    struct scsi_lun *lunp;
    ....

    /* Get the device number */
    dev = bp->b_dev;

    /* Get the pointer to scsi_bus structure */
    busp = m_scsi_bus(dev);

    /* Get the pointer to scsi_tgt structure */
    tgtp = m_scsi_tgt(dev);

    /* Get the pointer to scsi_lun structure */
    lunp = m_scsi_lun(dev);

    if(busp == NULL) {
        msg_printf("mydriver - a NULL scsi_bus
        pointer\n");
        return (ENXIO);
    }

    ....
}

SEE ALSO
NAME

scb(SCSI3) – SCSI Control Block Structure

SYNOPSIS

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

DESCRIPTION

SCSI services allocate `scb` structure and associate with a `buf` structure. The fields in the `scb` structure hold temporary state information until an I/O is completed.

This structure is used by an interface driver to get the SCSI command to be issued to the HBA it controls and to report the I/O completion status to the SCSI subsystem. The `scb` structure also has fields to pass any sense data that an interface driver may fill to the SCSI services layer.

When an interface driver calls `scsi_cbfn()` on completion of an I/O request, the SCSI services free the `scb` structure if the I/O is not going to be retried.

STRUCTURE MEMBERS

The following is a list of driver accessible fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void*</code></td>
<td><code>if_scb</code></td>
</tr>
<tr>
<td><code>struct scsi_lun *</code></td>
<td><code>lp</code></td>
</tr>
<tr>
<td><code>ubit32</code></td>
<td><code>flags</code></td>
</tr>
<tr>
<td><code>ubit32</code></td>
<td><code>max_msecs</code></td>
</tr>
<tr>
<td><code>ubit8</code></td>
<td><code>cdb</code></td>
</tr>
<tr>
<td><code>ubit8</code></td>
<td><code>cdb_len</code></td>
</tr>
<tr>
<td><code>ubit32</code></td>
<td><code>io_id</code></td>
</tr>
<tr>
<td><code>ubit8</code></td>
<td><code>tag</code></td>
</tr>
</tbody>
</table>
if_scb  

if_scb is a pointer to ifsw->if_scb_size bytes allocated by SCSI services and reserved for use by the interface driver. The pointer is initialized at scb creation time by services and the data area is bzero’ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time. The if_scb area is later freed by the SCSI services along with the scb.

lp  

lp is a pointer to the scsi_lun structure in the open device tree with which this scb is associated. If the scb belongs to a per-lun pool of scb’s as opposed to a per-bus pool, then scb->lp is initialized at scb creation time, by SCSI services, and never changed. Otherwise, the scb belongs to a per-bus pool, and scp->lp is only valid while the scb is associated with a bp. Then scb->lp points to the scsi_lun structure associated with bp->b_dev.

flags  

The interface driver may check these bits in the flag for proper functionality

<table>
<thead>
<tr>
<th>Flag</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCB_NO_DISC</td>
<td>This bit indicates that the disconnect privilege should not be granted in the identify message.</td>
</tr>
</tbody>
</table>

Table 7-1 Relevant scb Structure Fields (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubit32</td>
<td>cdb_status</td>
</tr>
<tr>
<td>ubit32</td>
<td>data_resid</td>
</tr>
<tr>
<td>ubit32</td>
<td>sense_status</td>
</tr>
<tr>
<td>ubit8</td>
<td>sense_bytes</td>
</tr>
<tr>
<td>ubit8*</td>
<td>sense_data</td>
</tr>
</tbody>
</table>
SCB_SDTR  If this bit is set and SCB_WDTR is not set, the interface driver should initiate SDTR negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending the CDB for the I/O.

SCB_WDTR  This bit directs the interface driver that a wide negotiation should be initiated immediately following the Selection, Identify or tag message, whichever comes last, and before sending any CDB for the I/O. If (tp->state & T_ENABLE_SDTR) or (scb->flags & SCB_SDTR) is also set, the interface driver should initiate SDTR negotiation immediately following the WDTR negotiation. The wide negotiation should always precede the synchronous negotiation, since a wide negotiation resets the link to asynchronous.

SCB_4BYTE  This bit is a hint to the interface driver that the target will never change phase while in data phase on other than a four-byte boundary (at the beginning of the data transfer) without subsequently restoring the data pointer (implicitly or explicitly) to a previously aligned value and re-transferring data up to and beyond the point of disconnection to an aligned boundary. The phase change at the end of the I/O need be considered only if the amount of data transferred may be less than that requested in bp->b_bcount. Note that SCB_4BYTE does not imply that bp->b_count is a multiple of four or that bp->b_un.b_addr is four-byte aligned. Note also that the phase
change out of data phase if all bp->b_bcount bytes have been transferred is not subject to the alignment restructuring.

**SCB_2BYTE**
This bit is the same as SCB_4BYTE except that phase changes are only restricted to even boundaries.

**SCB_ORDERED_TAG**
Denotes that ordered tags are intended to be used for this device.

**max_msecs**
Minimum number of milliseconds the interface driver is to allow for this I/O from the time of Selection until Command Complete. If scb->max_msecs milliseconds elapses and the I/O has not completed, the interface driver is encouraged to abort the I/O with Abort or Abort Tag as appropriate. A value of zero indicates the interface driver should never abort this I/O based solely on the amount of time since Selection.

**cdb**
Holds the SCSI command bytes for this I/O.

**cdb_len**
The number of bytes in the cdb. This can be a maximum of SCSI_MAX_CDB_LEN.

**io_id**
A unique identifier for a SCSI I/O. It is initialized when the scb is associated with a bp and is unique across all SCSI buses.

**tag**
The tag value allocated for this I/O by the SCSI subsystem in accordance with the interface driver's direction via ifsw->if_max_tag. It is recommended that the interface driver use this value as the tag value for the I/O if the I/O will be tagged, but it is not required. Currently there can only be 256 tags per bus. The tag value may not remain the same for retried I/Os.

**cdb_status**
Indicates the status of the I/O command. If the I/O attempt completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets scb->cdb_status to S_GOOD. If the selection phase times out, the interface driver sets cdb_status to SCTL_SELECT_TIMEOUT. If the I/O is not
even attempted because of bogus data in the bp or scb, the interface driver sets cdb_status to SCTL_INVALID_REQUEST. If the I/O is not attempted or does not complete for any other reason, cdb_status is set to SCTL_INCOMPLETE. If there is a Contingent Allegiance condition, the cdb_status is set to S_CHECK_CONDITION to request an auto-sense request. scb->cdb_status must be set by the interface driver prior to returning the bp via scsi_cbfn. Refer to scsi.h for all the valid values for the cdb_status.

data_resid  If the I/O attempt completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets scb->data_resid such that bp->b_count - scb->data_resid is the offset from bp->b_un.b_addr of the first byte not transferred by the target, i.e., number of bytes transferred = bp->b_bcount - scb->data_resid. Even if the I/O attempt is failed for some reason, it is advisable to set the scb->data_resid to indicate the number of bytes not yet transferred. Setting this field will have no adverse affect. scb->data_resid must be set by the interface driver prior to returning the bp via scsi_cbfn.

sense_status Represents the status of the automatic request sense that is performed if scb->cdb_status is S_CHECK_CONDITION. If the Request Sense completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets scb->sense_status. Otherwise, scb->sense_status is undefined and will not be referenced by the SCSI subsystem on callback. The possible values for scb->sense_status are the same as those for scb->cdb_status except SCTL_INVALID_REQUEST cannot be used. scb->sense_status represents the result of the automatic Request Sense in the same way that scb->cdb_status represents the result of attempting scb->cdb. It must be set by the interface driver before returning the bp via scsi_cbfn. If there is any sense data, the sense_status has to be set to S_GOOD.
sense_bytes  Number of bytes of data received in response to the automatic request sense if one was performed. It is valid only if sense_status is valid and is neither SCTL_SELECT_TIMEOUT nor SCTL_INCOMPLETE. scb->sense_bytes is the offset from scb->sense_data of the first byte of sense data not transferred by the target. It must be set by the interface driver prior to returning the bp via scsi_cbfn.

sense_data  If scb->cdb_status is Check Condition and the resulting Request Sense completes with no phase sequencing errors and without being aborted or timing out, and if scb->sense_status is not zero, the interface driver sets scb->sense_data. Otherwise, scb->sense_data is undefined and will not be referenced by the SCSI subsystem on callback. The interface driver sets scb->sense_data to point to a KERNELSPACE buffer containing the sense data; its size must be at least scb->sense_bytes. It must be set prior to returning the bp via scsi_cbfn and the interface driver must not modify the buffer for the duration of scsi_cbfn. When scsi_cbfn returns, and not until, the interface driver can reuse the buffer.

SEE ALSO

buf(KER4), scsi_ifsw(SCSI3)
NAME

scsi_action(SCSI3) – Give I/O completion information to SCSI Services

SYNOPSIS

#include <wsio/scsi_ctl.h>

int scsi_action (struct buf * bp, int flags, int error, int msecs);

PARAMETERS

bp Pointer to a buf structure
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 nontagged operation for the device. This is used to recover from tagged queueing 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() function 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.

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

-1  Do not retry. Call biodone()
SEE ALSO

biodone(KER2), panic(KER2)
NAME

`scsi_bus_lock(SCSI3)` – Acquire SCSI bus lock.

SYNOPSIS

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

void scsi_bus_lock(struct scsi_bus * busp);
```

PARAMETERS

`busp` Pointer to the scsi_bus structure.

DESCRIPTION

The `scsi_bus_lock()` and `scsi_bus_unlock()` functions are used to provide exclusive access to the `scsi_bus` structure. Calls to manipulate the I/O requests queues maintained by the SCSI services are bounded by `scsi_bus_lock()` and `scsi_bus_unlock()`. This can be called under interrupt context.

RETURN VALUES

None.

CONSTRAINTS

Must not be called while holding a spinlock with lock order >= `SCSI_BUS_LOCK_ORDER`
**EXAMPLE**

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

static int
mydriver_enqueue(struct scsi_bus *busp, struct buf *bp,
        int where)
{
    ....

    /* Enqueue the I/O request bp to the select_q */
    scsi_bus_lock(busp);
    scsi_enqueue(&busp->select_q, bp);
    scsi_bus_unlock(busp);

    ....
}
```

**SEE ALSO**

`scsi_dequeue(SCSI3), scsi_dequeue_bp(SCSI3), scsi_enqueue(SCSI3)`
NAME

\texttt{scsi\_bus\_unlock}(SCSI3) – Release SCSI bus lock.

SYNOPSIS

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

void scsi_bus_unlock(struct scsi_bus * busp);
\end{verbatim}

PARAMETERS

\begin{itemize}
\item \textit{busp} \hfill Pointer to the scsi_bus structure.
\end{itemize}

DESCRIPTION

The \texttt{scsi_bus_lock()} and \texttt{scsi_bus_unlock()} functions are used to provide exclusive access to the \texttt{scsi_bus} structure. Calls to manipulate the I/O requests queues maintained by the SCSI services are bounded by \texttt{scsi_bus_lock()} and \texttt{scsi_bus_unlock()}.

RETURN VALUES

None.

CONSTRAINTS
EXAMPLE

#include <sys/scsi_ctl.h>

static int
mydriver_enqueue(struct scsi_bus *busp, struct buf * bp,
     int where)
{
    ....

    /* Enqueue the I/O request bp to the select_q */
    scsi_bus_lock(busp);
    scsi_enqueue(&busp->select_q, bp);
    scsi_bus_unlock(busp);

    ....
}

SEE ALSO

scsi_dequeue(SCSI3), scsi_dequeue_bp(SCSI3),
scsi_enqueue(SCSI3)
NAME

`scsi_cbfn` (SCSI3) – SCSI subsystem callback function.

SYNOPSIS

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

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

PARAMETERS

`bp` Buffer pointer to the I/O request that is completed.

DESCRIPTION

The `scsi_cbfn` function is called by the interface driver on I/O attempt completion. When the interface driver finishes with an I/O, it returns the I/O to the SCSI subsystem by calling `scsi_cbfn` with the bp as its sole argument.

The interface driver relinquishes all rights to access bp, scb and `*scb->if_scb` once it calls `scsi_cbfn()`. Of course, the bp may be reused later for another I/O, and similarly for the scb and `*scb->if_scb`, although they will not necessarily be related in subsequent I/O's.

If the interface driver has attached a sense buffer to `scb->sense_data`, the sense_data buffer must be valid till `scsi_cbfn()` returns. The interface driver is forbidden from accessing it until `scsi_cbfn()` returns. It is important to note that the allocation and management of this buffer for holding sense_data is the responsibility of the interface driver.

This can be called either in process or interrupt context. This must not be called with any locks held.

RETURN VALUES

None

SEE ALSO

`scb` (SCSI3), `buf` (KER4)
NAME

scsi_cmd(SCSI3) – Prepare driver-generated I/O requests

SYNOPSIS

#include <wsio/scsi_ctl.h>

int scsi_cmd (dev_t dev, ubit32 flags, int cdb_len,
               ubit8 * cdb, int nbytes, void * addr,
               ubit32 msecs ubit32 retries, int * errp);

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.
errp 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.
RETURN 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

csci_cmdx(SCSI3) – Prepare driver-generated I/O requests

SYNOPSIS

#include <wsio/scsi_ctl.h>

int scsi_cmdx (dev_t dev, int flags, int cdb_len, u_char * cdb,
int nbytes, void * addr, u_int msecs,
u_int retries, int * errp,
struct status_action * sa, int n);

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.
errp If not NULL, then contains the error returned by the operation (in bp->b_error)
sa If NULL, there is no action to match or take.
n Status count.

DESCRIPTION

The scsi_cmdx() SCSI function is used for driver-generated I/O requests. It creates and builds a sctl_io and a bp, attaches the sctl_io 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

- **N** Number of bytes transferred.
- **-1** Error.

### SEE ALSO

`biowait(KER2), scsi_ctl(7), scsi_init_inquiry_data(SCSI3), scsi_strategy(SCSI3)`
NAME

scsi_ddsw(SCSI4) – SCSI device switch structure

SYNOPSIS

#include <wsio/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

blk_major        Obsolete field, not initialized.
raw_major        Obsolete field, not initialized.
dd_lun_size      The number of bytes to be allocated and attached to the open device tree when driver_open() is first executed.
dd_open()        Pointer to driver supplied routine.
dd_close()       Pointer to driver supplied routine.
dd_strategy()    Pointer to driver supplied routine.
dd_read()        Pointer to driver supplied routine.
dd_write()  Pointer to driver supplied routine.
dd_ioctl()  Pointer to driver supplied routine.
dd_start()  Pointer to driver supplied routine.
dd_done()  Pointer to driver supplied routine.

dd_pass_thru_okay()  Pointer to driver supplied routine.
dd_pass_thru_done()  Pointer to driver supplied routine.
dd_ioctl_okay()  Pointer to driver supplied routine.
dd_flags  Flag bits, currently only DD/DDG defined.

dd_status_list  Table of device driver status/action pairs.
dd_status_cnt  Count of device driver status/action pairs in the table.
wsio_drv  A pointer to your drivers wsio_drv_info structure.

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 customizing 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.

```c
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.

```c
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

\texttt{scsi\_dequeue} (SCSI3) – Remove I/O requests from queues maintained by SCSI
Services

SYNOPSIS

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

struct buf * scsi_dequeue (struct ** qp, int where);
\end{verbatim}

PARAMETERS

\begin{itemize}
\item \texttt{qp} Pointer to the head of a list of I/O requests.
\item \texttt{where} Location to extract from.
\end{itemize}

DESCRIPTION

The \texttt{scsi\_dequeue()} function extracts the I/O request at HEAD or TAIL
of the list. Parameter \texttt{*qp} is based on the value of \texttt{where} and return the
bp. This returns NULL when the queue is empty.

RETURN VALUES

\begin{itemize}
\item \texttt{NULL} Error.
\item \texttt{<>0} Pointer to struct \texttt{buf} (I/O request).
\end{itemize}

CONSTRAINTS

This must be called with \texttt{scsi\_bus} lock held.
EXAMPLE

#include <sys/scsi_ctl.h>

static int
mydriver_if_start(struct isc_table_type * isc)
{
    ....
    struct scsi_bus *busp;
    struct buf *bp;
    ....
    /* Get the SCSI bus pointer */
    busp = (struct scsi_bus *) isc->if_drv_data;

    /* Dequeue an I/O request from HEAD of the select_q */
    scsi_bus_lock(busp);
    bp = scsi_dequeue(&busp->select_q, HEAD);
    scsi_bus_unlock(busp);

    ....
}

SEE ALSO

scsi_dequeue_bp(SCSI3), scsi_enqueue(SCSI3)
NAME

scsi_dequeue_bp(SCSI3) – Remove a specific I/O request from a specified queue maintained by SCSI services.

SYNOPSIS

#include <sys/scsi_ctl.h>

struct buf * scsi_dequeue_bp(struct ** qp, struct buf * bp);

PARAMETERS

qp  Pointer to the head of a list of I/O requests.
bp  Specific buf to remove from the list.

DESCRIPTION

The scsi_dequeue_bp() function tries to dequeue bp from wherever it may be in the queue *qp. Returns bp when found on the queue. Returns NULL when not found on the queue.

RETURN VALUES

NULL    Error.
<>NULL   Pointer to struct buf (I/O request).

CONSTRAINTS

This must be called with scsi_bus lock held.
EXAMPLE

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

static int mydriver_dequeue_bp(struct scsi_bus *busp, struct buf * bp) {
    ....
    struct buf *rhp;
    ....
    /* Dequeue the I/O request bp from the select_q */
    scsi_bus_lock(busp);
    rbp = scsi_dequeue_bp(&busp->select_q, bp);
    scsi_bus_unlock(busp);
    ....
}
```

SEE ALSO

scsi_dequeue(SCSI3), scsi_enqueue(SCSI3)
NAME

_scsi_enqueue_(SCSI3) – Add buffer bp to a specified queue maintained by SCSI services.

SYNOPSIS

#include <sys/scsi_ctl.h>

void scsi_enqueue (struct buf **qp, struct buf *bp, int where);

PARAMETERS

qp A pointer to the head of a list of I/O requests.
bp Pointer to struct buf to be added to the list
where Location to insert the I/O request.

DESCRIPTION

The _scsi_enqueue_ function enqueues bp at the HEAD or TAIL of an I/O requests list; qp is a pointer to the list header which is a pointer to the head of the list. If _where_ is HEAD, the _bp_ is inserted ahead of the list, otherwise it is added to the tail of the list. There are different linked lists maintained by SCSI services: nexus_q, scb free lists, retry list, tag_q and busp->select_q.

RETURN VALUES

None.

CONSTRAINTS

This must be called with _scsi_bus_ lock held.
EXAMPLE

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

static int
mydriver_enqueue(struct scsi_bus *busp, struct buf * bp,
     int where)
{
    ....

    /* Enqueue the I/O request bp to the select_q */
    scsi_bus_lock(busp);
    scsi_enqueue(&busp->select_q, bp);
    scsi_bus_unlock(busp);

    ....
}
```

SEE ALSO

`scsi_dequeue(SCSI3), scsi_dequeue_bp(SCSI3)`
NAME

scsi_ifsw(SCSI3) – SCSI interface driver switch structure

SYNOPSIS

#include <sys/scsi_ctl.h>

DESCRIPTION

The interface driver switch structure exports an interface driver’s entry points and operational parameters to the SCSI services layer. The interface driver’s attach routine must initialize the ifsw field of the isc_table_type entry to point to a scsi_isw structure.

STRUCTURE MEMBERS

The scb structure is defined in <sys/scsi_ctl.h. The following are some important fields in the scb structure. Their types are given in the following table. Care should be taken when an interface driver modifies some of these fields.

The following is a list of driver accessible fields:

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubit8</td>
<td>if_flags;</td>
</tr>
<tr>
<td>ubit8</td>
<td>if_max_tag;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>if_scb_size;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>if_lun_size;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>if_tgt_size;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>if_bus_size;</td>
</tr>
<tr>
<td>int (*) ()</td>
<td>if_open</td>
</tr>
<tr>
<td>void (*) ()</td>
<td>if_close</td>
</tr>
<tr>
<td>void (*) ()</td>
<td>if_start</td>
</tr>
</tbody>
</table>
Interface driver flags convey information to the SCSI services on what it supports and what not. The possible flags are:

<table>
<thead>
<tr>
<th>Type Field Name</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF_BUS_TAGS</strong></td>
<td>A default flag.</td>
</tr>
<tr>
<td><strong>IF_NO_TAGS</strong></td>
<td>The interface driver does not support tags</td>
</tr>
<tr>
<td><strong>IF_B2_LIST</strong></td>
<td>The interface driver supports handling of disksort merge buffers</td>
</tr>
<tr>
<td><strong>IF_OWNS_TAGS</strong></td>
<td>The interface driver owns tagged queueing</td>
</tr>
</tbody>
</table>

One less than the number of per-bus tags supported by the interface driver. A tag is used to differentiate I/O requests. The SCSI subsystem will use tags from zero through `ifsw->if_max_tag`, inclusive. Actually, the interface driver is not required to use the tags allocated by the SCSI subsystem, but the SCSI subsystem will not allow more than `ifsw->if_max_tag+1` active I/O's to the bus at any given time (this includes untagged I/O's). Currently the maximum value of a tag can be 254.
if_scb_size  The number of bytes the SCSI subsystem shall allocate and attach to each scb for use by the interface driver. The if_scb field of scb structure is initialized at scb creation time by services and the data area is bzero'ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time.

if_lun_size  The number of bytes the SCSI subsystem shall allocate and attach to each scsi_lun structure for use by the interface driver. The if_lun field of scsi_lun structure is a pointer to ifsw->if_lun_size bytes for the use of interface driver.

if_tgt_size  The number of bytes the SCSI subsystem shall allocate and attach to each scsi_tgt structure for use by the interface driver. The if_tgt field of scsi_tgt structure is a pointer to ifsw->if_tgt_size bytes for the use of interface driver.

if_bus_size  The number of bytes the SCSI subsystem shall allocate and attach to each scsi_bus structure for use by the interface driver. The if_bus field of scsi_bus structure is a pointer to ifsw

if_open  Pointer to the interface driver’s logical unit open function. This is optional for an interface driver.

if_close  Pointer to the interface driver’s logical unit close function. This is optional for an interface driver.

if_start  Pointer to the interface driver’s start function.

if_reset_bus  Pointer to the interface driver’s Bus Reset function. This is optional for an interface driver.

if_bdr  Pointer to the interface driver’s Bus Device Reset function. This is optional for an interface driver.

if_abort  Pointer to the interface driver’s Abort function. This is optional for an interface driver.

if_io_max_size  Maximum size of I/O request supported by the interface driver. A value of 0 specifies no limit. If set, I/O requests for more than the supported size will be error’ed back by the SCSI services.
if_beg_align, if_end_align
Interface driver data buffer alignment requirement. These fields must be set to \((n - 1)\) where \(n\) is a power of two. SCSI services will ensure the data buffer \((bp->b_un.b_addr)\) is \(n\)-byte aligned. The maximum of both fields is used for buffer alignment.

SEE ALSO

buf(KER4), isc_table_type(KER4), scb(SCSI3)
NAME

`scsi_init_inquiry_data(SCSI3)` – Perform the first Inquiry request on a device

SYNOPSIS

```c
#include <wsio/scsi_ctl.h>

int scsi_init_inquiry_data (dev_t dev);
```

PARAMETERS

- `dev` Device number

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

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

scsi_ioctl(SCSI3) – Standard SCSI ioctl routine

SYNOPSIS

#include <wsio/scsi_ctl.h>

int scsi_ioctl (dev_t dev, int cmd, caddr_t data, int flags);

PARAMETERS

dev Device number of the associate device
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
flags The file access flags

DESCRIPTION

The scsi_ioctl() SCSI routine simplifies the job of the device driver. Ioctls which are supported by all device drivers are implemented here to insure consistency from one driver to the next, and to minimize maintenance costs.

Other ioctl commands may be supported by a particular driver's dd_ioctl() routine, which is invoked if the command is one that scsi_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.

scsi_ioctl() supports the following ioctl commands (defined in the <sys/scsi.h> header file):
DIOC_CAPACITY

`ioctl(fd, DIOC_CAPACITY, &capacity)`

Returns device size in `DEV_BSIZE` (1024) blocks. Returns information from data saved in LUN structure during the open. The structure `capacity` is defined in `../sys/diskio.h`.

DIOC_DESCRIBE

`ioctl(fd, DIOC_DESCRIBE, &describe)`

Returns information about the device. The `flags` field within the `describe_type` structure contains a write-protect flag for detection of physical write-protection on MO and WORM media. If the size of the describe data is not 32 bytes, EINVAL is returned. The returns information from data is saved in LUN structure during the open. The structure `describe` is defined in `../sys/diskio.h`.

DIOC_GET_PFTIMEOUT

`ioctl(fd, DIOC_GET_PFTIMEOUT, &msec)`

Returns integer value (`msecs`) used for timing all LVM requests within `bp->b_flags&B_PFTIMEOUT`. The structure `msec` is of type INT and defined in `../sys/diskio.h`.

DIOC_RSTCLR

`ioctl(fd, DIOC_RSTCLR)`

Perform reset on device (Bus Device Reset for SCSI). Defined in the `../sys/diskio.h`.

DIOC_SET_PFTIMEOUT

`ioctl(fd, DIOC_SET_PFTIMEOUT, &msecs)`

Sets integer value (`msecs`) used for timing all LVM requests within `bp->bflags&B_PFTIMEOUT`; zero (0) means reset to driver’s default. The structure `msecs` is of the type INT and is defined in `../sys/diskio.h`. 
- **SIOC_CAPACITY**
  
  `ioctl(fd, SIOC_CAPACITY, &capacity)`

  Returns device media capacity and device block size information from data saved in LUN structure during the open. The structure `capacity` is defined in `../sys/scsi.h`.

- **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).

  **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_FORMAT**
  
  `ioctl(fd, SIOC_FORMAT, &sioc_format)`

  Format device media capacity and block size. Must be superuser and have write access permissions or EACCES returned. Must have exclusive access or EBUSY returned. The structure `sioc_format` is defined in `../sys/scsi.h`. 
- **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 `../sys/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 `../sys/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 the `../sys/scsi.h`. See `scsi_ctl(7)`.
  
  Error Returns: None.

- **SIOC_GET_LUN_PARMS**
  
  `ioctl(fd, SIOC_GET_LUN_PARMS, &sioc_lun_parms)`
  
  The structure `sioc_lun_parms` is defined in `../sys/scsi.h`. See `scsi_ctl(7)`.
  
  Error Returns: None
SCSI Reference Pages

scsi_ioctl(SCSI3)

- **SIOC_GET_TGT_LIMITS**
  
  ```c
  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 `../sys/scsi.h`. See `scsi_ctl(7)`.

  Error Returns: None

- **SIOC_GET_TGT_PARMS**
  
  ```c
  ioctl(fd, SIOC_GET_TGT_PARMS, &sioc_tgtParms)
  ```

  The structure `sioc_tgtParms` is defined in `../sys/scsi.h`. See `scsi_ctl(7)`.

  Error Returns: None

- **SIOC_INQUIRY**
  
  ```c
  ioctl(fd, SIOC_INQUIRY, &inquiry_data)
  ```

  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 `../sys/scsi.h`.

- **SIOC_EXCLUSIVE**
  
  ```c
  ioctl(fd, SIOC_EXCLUSIVE, &int)
  ```

  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_IO
  
ioctl(fd, SIOC_IO, &sctl_io)

  This is used for pass-through I/Os.

  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:

  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.

  Parameter comments:

  The flags bit, `SCTL_READ`, specifies that the command is expected to have a data-in phase. If `data_length` is greater than zero and `SCTL_READ` is not set, the command is expected to have a data-out phase. `SCTL_INIT_WDTR` specifies the SCSI wide data transfer negotiation should be initiated. `SCTL_INIT_SDTR` specifies that SCSI synchronous data transfer negotiation should be initiated. The status values are the same as those for `SIOC_RETURN_STATUS`.

  There are several bits in `sctl->flags` that are interesting to the Interface Driver. `SCTL_NO_ATN` directs the Interface Driver to not assert ATN on selection for the I/O. If the target requests a message out anyway, the Interface Driver should send a No Op.
SCTL_INIT_WDTR directs the Interface Driver to initiate WDTR negotiation immediately following the Selection, identity or tag message, whichever comes last, and before sending the cdb for the I/O. If (tp->state & T_ENABLE_SDTR) or (scb->flags & SCB_INIT_SDTR) is also set, the Interface Driver should initiate SDTR negotiation immediately following the WDTR negotiation.

If SCTL_INIT_SDTR is set and SCTL_INIT_WDTR is not, the Interface Driver should initiate SDTR negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending the cdb for the I/O.

SCTL_4BYTE is a hint to the Interface Driver that the target will never change phase while in data phase on other than a 4-byte boundary (with regard to the beginning of the data transfer) without subsequently restoring the data pointer (implicitly or explicitly) to a previously aligned value and retransferring the data up to and beyond the point of the unaligned phase change. The Interface Driver is absolved from ensuring data integrity for the I/O if this bit is set and the above rule is broken. Of course, if the Interface Driver can detect the problem without impacting performance, it should indicate the failure to the SCSI subsystem. Note that SCTL_4BYTE does not imply that bp->b_bcount is a multiple of four (4) nor that bp->b_un.b_addr is 4-byte aligned. Note also that the phase change out of data phase if all bp->b_bcount bytes have been transferred is not subject to the alignment restriction.

SCTL_2BYTE is the same as SCB_4BYTE except phase changes are only restricted to even boundaries.

The following is defined in ../h/scsi.h

```c
struct sctl_io
{
    unsigned flags;        /* IN:SCTL_READ*/
    unsigned cdb_length;   /* IN*/
    unsigned char cdb[16]; /* IN*/
    void *date;            /* IN*/
    unsigned data_length;  /* IN*/
    unsigned max_msecs;    /* IN: multi-seconds before abort*/
    unsigned data_xfer;    /* OUT*/
    unsigned cdb_status;   /* OUT: SCSI status*/
    unsigned char sense[256]; /* OUT: SCSI status*/
    unsigned sense_status; /* OUT*/
    unsigned sense_xfer;   /* OUT: bytes of sense data received*/
    unsigned reserved[16]; /* IN: Must be zero; OUT: undefined*/
};
```
```c
#define   SIOC_IO   _IOWR('S', 22, struct sctl_io)

/*
** values for sctl_io->cdb_status and sctl_io->sense_status
*/
#define   SCTL_INVALID_REQUEST  0X0100
#define   SCTL_SELECT_TIMEOUT   0x0200
#define   SCTL_INCOMPLETE       0x0400

/*
** sctl_io->flags bits
*/
#define   SCTL_READ        0x00000001
#define   SCTL_INIT_WDTR   0x00000002
#define   SCTL_INIT_SDTR   0x00000004
#define   SCTL_NO_ATN      0x00000008 /*select without ATN, no SCSI messages*/
#define   SCTL_2BYTE       0x00000010 /*target maintains 2-byte alignment*/
#define   SCTL_4BYTE       0x00000020 /*target maintains 4-byte alignment*/

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 `[EACCES]`.  

  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_RESET_BUS**

  `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.
- **SIOC_RESET_DEV**

  `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.

- **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]`, or `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`).

  For SIOC_RETURN_STATUS ioctl, status in the `lun` structure utilizes an array of two 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;

    k = lp->cdb_status[i];
    lp->cdb_status[i] = -1;
}

* (u_int *) data = k;
return 0;
Error Returns: None

SIOC_SET_BUS_LIMITS

ioctl(fd, SIOC_SET_BUS_LIMITS, &sioc_bus_limits)

The structure sioc_bus_limits is defined in ../sys/scsi.h. See scsi_ctl (7).

Error Returns:

[EACCES] If not superuser or write permission.
[EINVAL] If reserved fields are not zero.
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 ../sys/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_SET_LUN_LIMITS

ioctl(fd, SIOC_SET_LUN_LIMITS, &sioc_lun_limits)

The structure sioc_lun_limits is defined in ../sys/scsi.h. See scsi_ctl (7).

Error Returns:

[EACCES] If not superuser or write permission.

[EINVAL] If reserved fields are not zero.

SIOC_SET_TGT_LIMITS

ioctl(fd, SIOC_SET_TGT_LIMITS, &sioc_tgt_limits)

The structure sioc_tgt_limits is defined in ../sys/scsi.h. See scsi_ctl (7).

Error Returns:

[EACCES] If not superuser or write permission.

[EINVAL] If reserved fields are not zero.
SIOC_XSENSE

`ioctl(fd, SIOC_XSENSE, &sense_data)`

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 two 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 become 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 if the request originates 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 `<scsi.h>` header file.

Error Returns:

`[EINVAL]` The data size is not equal to 128 bytes.

**RETURN VALUES**

0 Successful completion.

-1 Error.
NAME

scsi_lun_close(SCSI3) – Close a device

SYNOPSIS

void 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 non-pass-through close, wait for all non-pass-through I/O's to complete.
  - If this is any non-pass-through close, call device driver close routine, dd_close().
  - If this is the last non-pass-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 non-pass-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**

None
NAME

`scsi_lun_open(SCSI3)` – Open the elements of the hardware path of a SCSI lun

SYNOPSIS

```c
#include <wsio/scsi.ctl.h>

int scsi_lun_open (dev_t dev, struct scsi_ddsw *ddsw, 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) == scti_ddsw.raw_major`. 
RETURN 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.
[ENXIO] 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) – Read from device

SYNOPSIS

int scsi_read (dev_t dev, struct uio * uio);

PARAMETERS

dev 		Device number
uio 		Pointer to a uio structure.

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

0 	Successful completion.
<>0 	Error. The value is expected to be an errno value.

SEE ALSO

physio(KER2), scsi_ddsw(SCSI4)
NAME

\texttt{scsi\_sense\_action}(SCSI3) – Decode SCSI sense information

SYNOPSIS

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

scsi_sense_action (struct buf *bp,
                  struct sense_action *sense_list, size_t n)
\end{verbatim}

PARAMETERS

- \textit{bp} Pointer to the I/O buf structure
- \textit{sense\_list} List of actions to take.
- \textit{n} Number of actions in the list.

DESCRIPTION

The \texttt{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.

\texttt{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 \texttt{scsi\_init\_inquiry\_data()} before it can be interpreted. If an I/O completes before the inquiry data is initialized, \texttt{scsi\_sense\_action()} will not match anything other than wild card entries. It will panic if there is no matching entry.
RETURN VALUES

0     Successful completion.
<>0   Error. The value is provided by the sense action called.

SEE ALSO

scsi_init_inquiry_data(SCSI3)
NAME

scsi_strategy(SCSI3) – Enqueue the bp to await resources

SYNOPSIS

#include <wsio/scsi_ctl.h>

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  \texttt{scsi\_strategy()} \texttt{calls \texttt{dd\_strategy}()}, if present, holding the \texttt{lun\_lock}.

\section*{RETURN VALUES}

None.

\section*{SEE ALSO}

\texttt{biodone(KER2), scsi\_enqueue(SCSI3)}
NAME

`scsi_write(SCSI3)` – Write to device

SYNOPSIS

```c
#include <wsio/scsi_ctl.h>

int scsi_write (dev_t dev, struct uio * uiop);
```

PARAMETERS

- `dev` The device number
- `uiop` 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

- `0` Successful completion.
- `<>0` Error. The value is expected to be an `errno` value.

SEE ALSO

`physio(KER2), scsi_ddsw(SCSI4)`
A Deprecated WSIO Interfaces
This appendix contains WSIO reference pages that have been deprecated in HP-UX 11i. These interfaces are documented here to support their use in earlier versions. Developers are encouraged to use the newer WSIO interfaces.
Functions
NAME

init_map_context(CDIO3) – Macro to initialize mapping context structure

SYNOPSIS

#include <sys/dma.h>

#define init_map_context (map)

PARAMETERS

map 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 non-interleaving 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)
Deprecated WSIO Interfaces
Functions

NAME
isrlink(WSIO3) – Register an interrupt service routine

SYNOPSIS

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

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 ratline are enabled. The driver should be prepared to handle an interrupt from its device or another device sharing the irq_line.

RETURN VALUES

0       Successful completion.

WSIO_ERROR       Error.
CONSTRANTS

EXAMPLE

    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(WSIO).
Deprecated WSIO Interfaces
Functions

NAME

isrunlink(WSIO3) – Remove the ISR registered by isrlink()

SYNOPSIS

#include<wsio/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

0 Successful completion.
WSIO_ERROR Error.

CONSTRAINTS
SEE ALSO

isrlink(WSIO3)
Deprecated WSIO Interfaces
Functions

NAME

\texttt{m\_instance(WSIO3)} – Get the device instance field from the device number

SYNOPSIS

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

int m_instance(dev_t dev);
\end{verbatim}

PARAMETERS

\begin{verbatim}
dev
\end{verbatim}
Device number of the device.

DESCRIPTION

The \texttt{m\_instance()} WSIO macro returns the driver instance field from the device number.

RETURN VALUES

CONSTRAINTS

SEE ALSO
NAME

wsio_allocate_shared_memory (WSIO3) – Allocate and map contiguous memory used for continuous DMA.

SYNOPSIS

#include <wsio/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() WSIO function is a deprecated interface and may be obsoleted in a future release of HP-UX. Use the wsio_allocate_shared_mem() WSIO function in its place.

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_OPTIMIZE_DEVICE_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_OPTIMIZE_HOST_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**

- `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_OPTIMIZEDEVICE_LATENCY |
    WSIO_SHMEM_ALIGN_ON_SIZE) != SHMEM_OK) {

    /*
     * Failed to allocate memory for continuous DMA.
     */
}

SEE ALSO

wsio_allocate_shared_mem(WSIO3),
wsio_flush_shared_memory(WSIO3),
wsio_free_shared_memory(WSIO3)
Deprecated WSIO Interfaces
Functions

NAME
wsio_fastmap
(WSIO3) – Map all or part of a host address range into an I/O virtual address range

SYNOPSIS

```c
#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 (PC15) 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; \x11\x11\x11
    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`(WSIO3) – Flush the memory previously allocated and mapped by `wsio_allocate_shared_memory()`.

SYNOPSIS

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

shmem_status_t wsio_flush_shared_memory (  
    struct isc_table_type *isc, size_t size,  
    iova_t iova, vaddr_t vaddr,  
    wsio_shmem_attr_t type);
```

PARAMETERS

- `isc` Pointer to an ISC table entry.
- `size` Size in bytes or memory to be flushed.
- `iova` I/O virtual address.
- `vaddr` Virtual address corresponding to `iova`.
- `type` Bit mask of the memory attributes.

DESCRIPTION

The `wsio_flush_shared_memory()` WSIO function is a deprecated interface and may be obsoleted in a future release of HP-UX. Use the `wsio_flush_shared_mem()` WSIO function in its place.

The `wsio_flush_shared_memory()` WSIO function flushes the memory previously allocated and mapped by `wsio_allocate_shared_memory()`. This ensures that data in the allocated memory is viewed consistently by the device and processors. All parameters passed to `wsio_flush_shared_memory()` must match the parameters passed to the corresponding call to `wsio_allocate_shared_memory()`.

RETURN VALUES

- `SHMEM_OK` Successful completion
- `SHMEM_NO_RESOURCES` Memory not flushed
Deprecation WSIO Interfaces

Functions

CONSTRAINTS

SEE ALSO

wsio_allocate_shared_memory(WSIO3),
wsio_flush_shared_mem(WSIO3), wsio_free_shared_memory(WSIO3)
NAME

wsio_free_shared_memory(WSIO3) – Release and unmap contiguous memory previously allocated and mapped by wsio_allocate_shared_memory().

SYNOPSIS

#include <wsio/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() WSIO function is a deprecated interface and may be obsoleted in a future release of HP-UX. Use the wsio_free_shared_mem() WSIO function in its place.

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_shared_memory().

RETURN VALUES

None
deprecated WSIO Interfaces

Functions

CONTRAINTS

SEE ALSO

wsio_allocate_shared_mem(WSIO3), wsio_free_shared_mem(WSIO3),
wsio_flush_shared_mem(WSIO3)
NAME

wsio_get_interrupts (WSIO3) – Determine which interrupt has been assigned to a card

SYNOPSIS

#include <sys/wsio.h>

input_t 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

>0 The IRQ for the card.
-1 isc is NULL or there was a problem retrieving the node's interrupt property.

CONSTRAINTS
Deprecated WSIO Interfaces

Functions

EXAMPLES

```c
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_pva(WSIO3)` – Translate an IO virtual address to its processor virtual address

SYNOPSIS

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

void * 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 macro 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

Appendix A  617


#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/wsio.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
SEE ALSO
NAME

`wsio_install_drv_func` (WSIO_DRV) – Register a driver function with the WSIO driver environment.

SYNOPSIS

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

int wsio_install_drv_func (wsio_drv_info_t *drv_hdr,
                           wsio_drv_func_type_t func_id,
                           wsio_drv_func_t drv_func,
                           wsiouintptr_t arg1,
                           wsiouintptr_t arg2);
```

PARAMETERS

- `drv_hdr` A pointer to the driver's `wsio_drv_info_t` structure.
- `func_id` Identify what function a driver is registering.
- `drv_func` A driver's function to register with `wsio`.
- `arg1` `func_id` dependent.
- `arg2` `func_id` dependent.

**func_id**

- `WSIO_DRV_CLAIM_FUNC` A driver's claim function to claim underlying devices.
  - `arg1` & `arg2` - not applicable; set to 0.

DESCRIPTION

This function is used to register a driver function with WSIO. The type of function is identified by `func_id`. When a service is needed, WSIO will execute this function. Depending on the function type, the `args` may be passed back to a driver. A driver should call this function only after it has called `wsio_install_driver()`.
RETURN VALUES

WSIO_OK
   Successful completion.

WSIO_DRV_FUNC_NULL
   drv_func is NULL.

WSIO_DRV_NOT_FOUND
   Invalid driver. Driver should call
   wsio_install_driver() first.

WSIO_INFO_NULL
   wsio_drv_info_t pointer is NULL.

WSIO_UNKNOWN_FUNC_TYPE
   Invalid func_id.

CONSTRAINTS

EXAMPLE

static wsio_drv_info_t my_drv_info {
    ....
} int my_claim(wsio_generic_data_t *data_ptr)
{
    ............
}
int my_install(void)
{
    ....
    if (wsio_install_driver(&my_drv_info)) {
        ....
        /* Register my claim function, arg1 and arg2 are
         * reserved, set to 0
         */
        if (wsio_install_drv_func(&my_drv_info,
                                   WSIO_DRV_CLAIM_FUNC, my_claim, 0, 0))
            ....
    }
    ....
}
SEE ALSO

wsio_install_drv_event_handler(WSIO3),
wsio_query_supported_function(WSIO_DRV)
NAME

wsio_map(WSIO3) – Map all or part of a host address range into an I/O virtual address range

SYNOPSIS

#include <sys/wsio.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.
Deprecated WSIO Interfaces

Functions

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, this is the space ID of the address range. Other values are:

KERNELSPACE Kernel virtual buffer (same as space ID = 0)
**Deprecated WSIO Interfaces**

**Functions**

- **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()` macro 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). Buffers 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 buffers 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

>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_NO_SEQ flag bits set to ensure coherency. See pci-errata (PC15).

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;

sample_pci_attach(id, isc)
PCI_ID id;
struct isc_table_type *isc;
{
    caddr_t tmp_buf,host_ram;
Deprecated WSIO Interfaces

Functions

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
** 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 
G) {
        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_savedAttach)(idc, isc);
```

```
Deprecated WSIO Interfaces

Functions

```c
}
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)
Deprecated WSIO Interfaces
Functions

NAME

wsio_register_probe_func(WSIO3) – Insert a driver-specified probe function into
the global probe list

SYNOPSIS

#include<sys/wsio.h>

void 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_t structs registered
during driver_install().

DESCRIPTION

The WSIO service wsio_register_probe_func() 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_t struct.

RETURN VALUES

None.

CONSTRAINTS
SEE ALSO

driver_class_probe(WSIO_DRV), driver_addr_probe(WSIO_DRV),
wsio_drv_data_t(WSIO4)
Deprecated WSIO Interfaces

Functions

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**

- 0 The range was fully mapped.
- -1 The necessary resources could not be obtained.

**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.
Deprecated WSIO Interfaces

Functions

IO_NONINTERLEAVED_MEMORY
For small buffers that won’t cross a page boundary, and page size in length.

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)`
Deprecated WSIO Interfaces

Functions

NAME

wsio_unmap (WSIO3) – Unmap an I/O virtual address range

SYNOPSIS

```c
#include <sys/wsio.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 `wsio_unmap()` the caller must call `dma_sync_IO()` 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

None.

CONSTRAINTS
EXAMPLE

The following function cleans up after an inbound DMA:

```c
#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(KERNELSPACE,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)
Deprecated WSIO Interfaces

Functions
Index

ETHER_MAP_IP_MULTICAST Network
  macro, 410

F
FREE() kernel macro, 59
free_isc() WSIO function, 226
functions
  alloc_spinlock(), 22
  b_cpsema(), 24
  b_initsema(), 27
  b_owns_sema(), 29
  b_psema(), 31
  b_vsema(), 33
  bcmp(), 35
  bcopy(), 36
  beta semaphore, 31, 33
  biodone(), 38
  biowait(), 40
  bp_dma_cleanup(), 188
  bp_dma_setup(), 190
  brelse(), 42
  busywait(), 49
  bzero(), 50
  copyin(), 51
  copyout(), 53
  cspinlock(), 55
  dd_close(), 506
  dd_done(), 507
  dd_ioctl(), 508
  dd_ioctl_okay(), 509
  dd_open(), 510
  dd_pass_thru_done(), 511
  dd_pass_thru_okay(), 512
  dd_read(), 513
  dd_start(), 514
  dd_strategy(), 515
  dd_write(), 516
  dealloc Spinlock(), 58
  dma_cleanup(), 192
  dma_setup(), 195
  driver_addr_probe(), 197
  driver_attach(), 199
  driver_class_probe(), 204
  driver_close(), 201
  driver_dev_init(), 203
  driver_if_abort(), 517
  driver_if_bdr(), 518
  driver_if_close(), 519
  driver_if_init(), 206
  driver_if_open(), 520
  driver_if_reset_bus(), 521
  driver_if_start(), 523
  driver_install(), 207
  driver_ioctl(), 209
  driver_isr(), 211
  driver_minphys(), 212
  driver_open(), 216
  driver_psize(), 218
  driver_read(), 219
  driver_select(), 221
  driver_strategy(), 223
  driver_write(), 224
  free_isc(), 226
  get_new_isc(), 227
  get_sleep_lock(), 61
  getc(), 63
  getcb(), 65
  getcf(), 67
  geteblk(), 68
  geterror(), 70
  gsignal(), 72
  hw_ift_attach(), 416
  install_driver(), 183
  iodone(), 229
  iowait(), 231
  isc_claim(), 233
  isrunlink(), 600
  isrunlink(), 602
  iwsio_get_processor_count(), 299
  kernel_iomap(), 75
  kernel_iomap_public(), 78
  kernel_iounmap(), 81
  kget_log_instance(), 419
  klogg_write(), 422
  kmalloc(), 83
  Ktimeout(), 85
  ktrc_write(), 426
  ldsid(), 87
  m_lun_id(), 527
  m_scsi_bus(), 531
  m_scsi_isc(), 533
  m_scsi_lun(), 535
  m_scsi_tgt(), 537
  m_wsio_funcnum(), 239
  m_wsio_selcode(), 240
  m_wsio_vsc(), 241
  map_mem_to_host(), 94
Index

minphys(), 97
mod_wsio_attach_list_add(), 242
mod_wsio_attach_list_remove(), 244
msg_printf(), 99
owns_spinlock(), 101
p_pgrp(), 103
panic(), 105
pci_desc_bus_transactions_isc(), 475
pci_get_fru_info_isc(), 477
pci_get_port_hndl_isc(), 479
pci_read_cfg_uintN_isc(), 481
pci_read_port_uintN_isc(), 483
pci_unget_port_hndl_isc(), 486
pci_write_cfg_uintN_isc(), 488
pci_write_port_uintN_isc(), 490
physio(), 106
printf(), 110
privlbcopy(), 112
psignal(), 114
putc(), 116
putcb(), 118
putcf(), 119
scsi_action(), 545
scsi_bus_lock(), 548
scsi_bus_unlock(), 550
scsi_chfn(), 552
scsi_cmd(), 553
scsi_cmds(), 555
scsi_dequeue(), 561
scsi_dequeue_bpt(), 563
scsi_enqueue(), 565
scsi_init_inquiry_data(), 571
scsi_ioctl(), 572
scsi_lun_close(), 585
scsi_lun_open(), 587
scsi_read(), 589
scsi_sense_action(), 590
scsi_strategy(), 592
scsi_write(), 594
seltrue(), 121
selwakeup(), 123
set_up_8022(), 428
set_up_ether(), 430
set_up_ip(), 432
set_up_link(), 434
sleep(), 125
spinlock(), 58, 61, 128
spinunlock(), 130
printf(), 132
streat(), 134
strcmp(), 135
strcpy(), 137
strlen(), 139
strncmp(), 141
strncpy(), 143
subsys_N_format(), 436
subsys_N_get_options(), 443
suser(), 145
sw_trigger(), 146
timeout(), 149
tl_banner_char(), 447
tl_check_cat_version(), 448
tl_format_fprintf(), 450
tl_format_write(), 453
tl_get_line(), 456
tl_get_parms(), 458
tl_header_format1(), 462
tl_log_class(), 465
tl_rae_format(), 467
tl_trace_kind(), 469
uiomove(), 153
unmap_mem_from_host(), 156
untimeout(), 157
user_iomap(), 158
user_iounmap(), 161
VASSERT(), 162
waiting_in_select(), 163
wakeup(), 165
wsio_activate_probe(), 246
wsio_alloc_mem(), 249
wsio_alloc_mem_handle(), 251
wsio_allocat_shared_memory(), 605
WSIO_BIG_ENDIAN(), 260
wsio_dma_pass_thru(), 266
wsio_dma_set_device_attributes(), 269
wsio_fastmap(), 608
wsio_fastmap_dma_buffer(), 279
wsio_flush_shared_mem(), 282
wsio_flush_shared_memory(), 611
wsio_free_dma_handle(), 284
wsio_free_mem(), 286
wsio_free_mem_handle(), 287
wsio_free_shared_mem(), 288
wsio_free_shared_memory(), 613
wsio_get_active_processor_count(), 292
wsio_get_all_registers(), 293
wsio_get_interrupts(), 615
wsio_get_ioports(), 295
Index

wsio_get_isc(), 297
wsio_get_pva(), 617
wsio_get_registers(), 620
wsio_get_system_params(), 300
wsio_install_driver(), 304
wsio_install_drv_event_handler(), 307
wsio_install_drv_func(), 622
wsio_intr_activate(), 309
wsio_intr_alloc(), 312
wsio_intr_deactivate(), 314
wsio_intr_deactivate_nowait(), 317
wsio_intr_free(), 320
wsio_intr_get_assigned_cpu(), 322
wsio_intr_get_irq_line(), 324
wsio_intr_get_txn_info(), 326
wsio_intr_set_cpu_spec(), 332
wsio_intr_set_irq_line(), 336
wsio_map(), 625
wsio_map_cfg_handle(), 345
wsio_map_dma_buffer(), 347
wsio_map_port(), 353
wsio_map_reg(), 355
wsio_port_inXX(), 359
wsio_port_outXX(), 361
wsio_query_supported_function(), 365
wsio_read_regXX(), 367
wsio_reg_drv_capability_mask(), 369
wsio_register_addr_probe(), 371
wsio_register_dev_probe(), 374
wsio_register_probe_func(), 634
wsio_remap(), 377
wsio_remap_dma_buffer(), 377
wsio_set_attributes(), 638
wsio_set_description(), 381
wsio_set_dma_attributes(), 383
wsio_set_dma_callback(), 388
wsio_uninstall_driver(), 390
wsio_uninstall_drv_event_handler(), 391
wsio_unmap(), 640
wsio_unmap_dma_buffer(), 395
wsio_unmap_port(), 398
wsio_unmap_reg(), 400
wsio_unregister_dev_probe(), 402
wsio_write_regXX(), 405

G
get register addresses of interface card, 620
get_new_isc() WSIO function, 227
get_sleep_lock() kernel function, 61
getc() kernel function, 63
getcb() kernel function, 65
getc() kernel function, 67
getbli() kernel function, 68
geterror() kernel function, 70
gsignal() kernel function, 72

H
hw_ift() Network structure, 411
hw_ift_attach() Network function, 416
hwIFT_LOCK() Network macros, 417
hwIFT_UNLOCK() Network macro, 417

I
init_map_context() CDIO macro, 598
install_driver() CDIO function, 183
instance number of iotree node, 341
intended audience, 15
interface card register addresses, 620
interface select code table, 235
interrupt assigned to card, 615
interrupt service routine, 211
intr_migr_info_t WSIO structure, 331
iodone() WSIO function, 229
iotree node instance number, 341
iovec kernel structure, 74
iowait() WSIO function, 231
isc_claim() WSIO function, 233
ISC structure element, claim, 233
ISC structure pointer for device file, 297
isc_claim() WSIO function, 233
isc_table_type WSIO structure, 235
isrlink() WSIO function, 600
isrunlink() WSIO function, 602

K
kernel functions
  alloc_spinlock(), 22
  b_cpsema(), 24
  b_initsema(), 27
  b_owns_sema(), 29
  b_psema(), 31
  b_vsema(), 33
  bcmp(), 35
  bcopy(), 36
  beta semaphore, 31, 33
  biodone(), 38
biowait(), 40
brelse(), 42
busywait(), 49
bzero(), 50
copyin(), 51
copyout(), 53
cspinlock(), 55
dealloc_spinlock(), 58
get_sleep_lock(), 61
getc(), 63
getcb(), 65
getcf(), 67
getblklk(), 68
gerror(), 70
gsignal(), 72
kernet_iomap(), 75
kernel_iomap_public(), 78
kernel_iounmap(), 81
kmalloc(), 83
Ktimeout(), 85
ldsid(), 87
map_mem_to_host(), 94
minphys(), 97
msg_printf(), 99
owns_spinlock(), 101
p_pgrp(), 103
panic(), 105
physio(), 106
printf(), 110
privlbcopy(), 112
psignal(), 114
putc(), 116
putcb(), 118
putcf(), 119
seltrue(), 121
selwakeup(), 123
sleep(), 125
spinlock(), 58, 61, 128
spinunlock(), 130
sprintf(), 132
strcat(), 134
strcmp(), 135
strcpy(), 137
strlen(), 139
strnclnp(), 141
strncpy(), 143
suser(), 145
sw_trigger(), 146
timeout(), 149
uiomove(), 153
unmap_mem_from_host(), 156
untimout(), 157
user_iomap(), 158
user_iounmap(), 161
VASSERT(), 162
waiting_in_select(), 163
wakeup(), 165
kernel macros
FREE(), 59
kfree(), 82
major(), 89
makedev(), 90
MALLOC(), 91
minor(), 96
kernel structures
buf, 43
iovec, 74
uio, 151
kernel_iomap() kernel function, 75
kernel_iomap_public() kernel function, 78
kernel_iounmap() kernel function, 81
kmalloc() kernel function, 83
Ktimeout() kernel function, 85
KTRC_CHECK() Network macro, 424
ktrc_write() Network function, 426
ldsid() kernel function, 87
m_bus_id() SCSI macro, 525
m_instance() WSIO macro, 604
m_lun_id() SCSI macro, 527
m_scsi_bus() SCSI function, 531
m_scsi_isc() SCSI function, 533
m_scsi_lun() SCSI function, 535
m_scsi_tgt() SCSI function, 537
m_tgt_id() SCSI macro, 529
m_wsio_funcnum() WSIO function, 239
m_wsio_selcode() WSIO function, 240
m_wsio_vsc() WSIO function, 241
macros
CONNECT_INIT_ROUTINE(), 473
dma_sync(), 170
dma_sync_IO(), 173
Index

ETHER_MAP_IP_MULTICAST, 410
FREE(), 59
HWIFT_LOCK(), 417
HWIFT_UNLOCK(), 417
init_map_context(), 598
kfree(), 82
KLOG_CK(), 420
KTRC_CHECK(), 424
m_bus_id(), 525
m_instance(), 604
m_tgt_id(), 529
major(), 89
makedev(), 90
MALLOC(), 91
minor(), 96
PCI_ATTACH_DEV_INIT_ERROR(), 474
READ_REG_UINTn_ISC(), 492
WRITE_REG_UINTn_ISC(), 495
wsio_allocate_dma_handle(), 254
wsio_allocate_shared_mem(), 256
wsio_cfg_inXX(), 262
wsio_cfg_outXX(), 264
wsio_init_map_context(), 302
WSIO_LITTLE_ENDIAN(), 343
WSIO_ORDERED_INTERRUPTS(), 357
wsio_unmap_cfg_handle(), 393
major() kernel Macro, 89
makedev() kernel Macro, 90
MALLOC() kernel macro, 91
manpage, topic format, 17
mark ISC structure element as claimed by
driver, 233
Memory Coherency Issues, 498
minor number, build, 212
minor() kernel macro, 96
minphys() kernel function, 97
mod_wsio_attach_list_add() WSIO function,
242
mod_wsio_attach_list_remove() WSIO
function, 244
msg_printf() kernel function, 99

N
Network functions
hw_ift_attach(), 416
kget_log_instance(), 419
klogg_write(), 422
ktrc_write(), 426
set_up_8022(), 428
set_up_ether(), 430
set_up_ip(), 432
set_up_link(), 434
subsys_N_format(), 436
subsys_N_get_options(), 443
tl_banner_char(), 447
tl_check_cat_version(), 448
tl_format_fprintf(), 450
tl_format_write(), 453
tl_get_line(), 456
tl_get_parms(), 458
tl_header_format1(), 462
tl_log_class(), 465
tl_rae_format(), 467
tl_trace_kind(), 469
Network macros
ETHER_MAP_IP_MULTICAST, 410
HWIFT_LOCK(), 417
HWIFT_UNLOCK(), 417
KLOG_CK(), 420
KTRC_CHECK(), 424
Network structures
hw_ift(), 411

O
owns_spinlock() kernel function, 101

P
p_pgrp() kernel function, 103
panic() kernel function, 105
PCI Config. Cycle Retry Problem, 502
PCI errata, 498, 501, 502
PCI functions
pci_desc_bus_transactions_isc(), 475
pci_get_fru_info_isc(), 477
pci_get_port_hndl_isc(), 479
pci_read_cfg_uintN_isc(), 481
pci_read_port_uintN_isc(), 483
pci_unget_port_hndl_isc(), 486
pci_write_cfg_uintN_isc(), 488
pci_write_port_uintN_isc(), 490
PCI macros
CONNECT_INIT_ROUTINE(), 473
PCI_ATTACH_DEV_INIT_ERROR(), 474
READ_REG_UINTn_ISC(), 492
WRITE_REG_UINTn_ISC(), 495
PCI Transaction Ordering Errata, 501, 502
PCI_ATTACH_DEV_INIT_ERROR() PCI
macro, 474
Index

pci_desc_bus_transactions_isc() PCI function, 475
pci_get_fru_info_isc() PCI function, 477
pci_get_port_hndl_isc() PCI function, 479
pci_read_cfg_uintN_isc() PCI functions, 481
pci_unget_port_hndl_isc() PCI functions, 483
pci_write_cfg_uintN_isc() PCI functions, 488
pci_write_port_uintN_isc() PCI functions, 490
physio() kernel function, 106
printf() kernel function, 110
privilbcopy() kernel function, 112
probe structure, device, 363
probing function, 197, 204
psignal() kernel function, 114
putc() kernel function, 116
putc() kernel function, 118
putc() kernel function, 119

R
READ_REG_UINTn_ISC() PCI macros, 492
register addresses of interface card, 620
retrieve instance number of iotree node, 341
retrieve ISC structure pointer for device file, 297
Retry Problem, Config. Cycle PCI, 502
routines
  character device read/write, 219, 224

S
scb() SCSI structure, 539
SCSI functions
  dd_close(), 506
  dd_done(), 507
  dd_ioctl(), 508
  dd_ioctl_okay(), 509
  dd_open(), 510
  dd_pass_thru_done(), 511
  dd_pass_thru_okay(), 512
  dd_read(), 513
  dd_start(), 514
  dd_strategy(), 515
  dd_write(), 516
  driver_if_abort(), 517
  driver_if_bdr(), 518
  driver_if_close(), 519
  driver_if_open(), 520
  driver_if_reset_bus(), 521
  driver_if_start(), 523
  m_scsi_bus(), 531
  m_scsi_isc(), 533
  m_scsi_lun(), 535
  m_scsi_tgt(), 537
  scsi_action(), 545
  scsi_bus_lock(), 548
  scsi_bus_unlock(), 550
  scsi_cbfn(), 552
  scsi_cmds(), 555
  scsi_dequeue(), 561
  scsi_dequeue_bp(), 563
  scsi_enqueue(), 565
  scsi_init_inquiry_data(), 571
  scsi_ioctl(), 572
  scsi_lun_close(), 585
  scsi_lun_open(), 587
  scsi_read(), 589
  scsi_sense_action(), 590
  scsi_strategy(), 592
  scsi_write(), 594
SCSI macros
  m_bus_id(), 525
  m_lun_id(), 527
  m_tgt_id(), 529
SCSI structures
  scb(), 539
  scsi_ddsw(), 557
  scsi_ifsw(), 567
  scsi_action() SCSI function, 545
  scsi_bus_lock() SCSI function, 548
  scsi_bus_unlock() SCSI function, 550
  scsi_cbfn() SCSI function, 552
  scsi_cmd() SCSI function, 553
  scsi_cmds() SCSI function, 555
  scsi_ddsw() SCSI structure, 557
  scsi_dequeue() SCSI function, 561
  scsi_dequeue_bp() SCSI function, 563
  scsi_enqueue() SCSI function, 565
  scsi_ifsw() SCSI structure, 567
  scsi_init_inquiry_data() SCSI function, 571
  scsi_ioctl() SCSI function, 573
  DIOC_CAPACITY, 573
  DIOC_DESCRIBE, 573
  DIOC_GET_PFTIMEOUT, 573
  DIOC_RSTCLR, 573
  DIOC_SET_PFTIMEOUT, 573
  SIOC_CAPACITY, 574
  SIOC_CMD_MODE, 574
  SIOC_FORMAT, 574
  SIOC_GET_BUS_LIMITS, 575
### Index

<table>
<thead>
<tr>
<th>Function/Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOC_GET_BUS_PARMS</td>
<td>575</td>
</tr>
<tr>
<td>SIOC_GET_LUN_PARMS</td>
<td>575</td>
</tr>
<tr>
<td>SIOC_GET_TGT_PARMS</td>
<td>576</td>
</tr>
<tr>
<td>SIOC_INQUIRY</td>
<td>576</td>
</tr>
<tr>
<td>SIOC_IO_MODE</td>
<td>577</td>
</tr>
<tr>
<td>SIOC_PRIORITY_MODE</td>
<td>580</td>
</tr>
<tr>
<td>SIOC_RESET_BUS</td>
<td>580</td>
</tr>
<tr>
<td>SIOC_RESET_DEV</td>
<td>581</td>
</tr>
<tr>
<td>SIOC_RESET_STATUS</td>
<td>581</td>
</tr>
<tr>
<td>SIOC_SET_BUS_LIMITS</td>
<td>582</td>
</tr>
<tr>
<td>SIOC_SET_DEV</td>
<td>581</td>
</tr>
<tr>
<td>SIOC_SET_LUN_LIMITS</td>
<td>582</td>
</tr>
<tr>
<td>SIOC_SET_LUN_PARMS</td>
<td>583</td>
</tr>
<tr>
<td>SIOC_SET_TGT_LIMITS</td>
<td>583</td>
</tr>
<tr>
<td>SIOC_SET_TGT_PARMS</td>
<td>583</td>
</tr>
<tr>
<td>SIOC_XSENSE</td>
<td>583, 584</td>
</tr>
<tr>
<td>scsi_ioctl()</td>
<td>572</td>
</tr>
<tr>
<td>scsi_lun_close()</td>
<td>585</td>
</tr>
<tr>
<td>scsi_lun_open()</td>
<td>587</td>
</tr>
<tr>
<td>scsi_read()</td>
<td>589</td>
</tr>
<tr>
<td>scsi_sense_action()</td>
<td>590</td>
</tr>
<tr>
<td>scsi_strategy()</td>
<td>592</td>
</tr>
<tr>
<td>scsi_write()</td>
<td>594</td>
</tr>
<tr>
<td>select code</td>
<td>235</td>
</tr>
<tr>
<td>seltrue()</td>
<td>121</td>
</tr>
<tr>
<td>selwakeup()</td>
<td>123</td>
</tr>
<tr>
<td>set_up_ether()</td>
<td>428</td>
</tr>
<tr>
<td>set_up_ip()</td>
<td>430</td>
</tr>
<tr>
<td>set_up_link()</td>
<td>432</td>
</tr>
<tr>
<td>SIOC_CAPACITY</td>
<td>574</td>
</tr>
<tr>
<td>SIOC_CMD_MODE</td>
<td>574</td>
</tr>
<tr>
<td>SIOC_FORMAT</td>
<td>574</td>
</tr>
<tr>
<td>SIOC_GET_BUS_LIMITS</td>
<td>575</td>
</tr>
<tr>
<td>SIOC_GET_BUS_PARMS</td>
<td>575</td>
</tr>
<tr>
<td>SIOC_GET_LUN_LIMITS</td>
<td>575</td>
</tr>
<tr>
<td>SIOC_GET_LUN_PARMS</td>
<td>575</td>
</tr>
<tr>
<td>SIOC_GET_TGT_LIMITS</td>
<td>576</td>
</tr>
<tr>
<td>SIOC_GET_TGT_PARMS</td>
<td>576</td>
</tr>
<tr>
<td>SIOC_INQUIRY</td>
<td>576</td>
</tr>
<tr>
<td>SIOC_IO_MODE</td>
<td>577</td>
</tr>
<tr>
<td>SIOC_PRIORITY_MODE</td>
<td>580</td>
</tr>
<tr>
<td>SIOC_RESET_BUS</td>
<td>580</td>
</tr>
<tr>
<td>SIOC_RESET_DEV</td>
<td>581</td>
</tr>
<tr>
<td>SIOC_RESET_STATUS</td>
<td>581</td>
</tr>
<tr>
<td>SIOC_SET_BUS_LIMITS</td>
<td>582</td>
</tr>
<tr>
<td>SIOC_SET_CMD</td>
<td>583</td>
</tr>
<tr>
<td>SIOC_SET_DEV</td>
<td>581</td>
</tr>
<tr>
<td>SIOC_SET_LUN_LIMITS</td>
<td>582</td>
</tr>
<tr>
<td>SIOC_SET_LUN_PARMS</td>
<td>583</td>
</tr>
<tr>
<td>SIOC_SET_TGT_LIMITS</td>
<td>583</td>
</tr>
<tr>
<td>SIOC_XSENSE</td>
<td>584</td>
</tr>
<tr>
<td>sleep()</td>
<td>125</td>
</tr>
<tr>
<td>sleep()</td>
<td>125</td>
</tr>
<tr>
<td>spinlock()</td>
<td>58, 61, 128</td>
</tr>
<tr>
<td>spinlock()</td>
<td>58, 128</td>
</tr>
<tr>
<td>spinunlock()</td>
<td>130</td>
</tr>
<tr>
<td>sprintf()</td>
<td>132</td>
</tr>
<tr>
<td>strcat()</td>
<td>134</td>
</tr>
<tr>
<td>strcmp()</td>
<td>135</td>
</tr>
<tr>
<td>strcpy()</td>
<td>137</td>
</tr>
<tr>
<td>strlen()</td>
<td>139</td>
</tr>
<tr>
<td>strncmp()</td>
<td>141</td>
</tr>
<tr>
<td>strncpy()</td>
<td>143</td>
</tr>
<tr>
<td>structures</td>
<td>235</td>
</tr>
<tr>
<td>buf</td>
<td>43</td>
</tr>
<tr>
<td>dma_parms</td>
<td>176</td>
</tr>
<tr>
<td>drv_info</td>
<td>179</td>
</tr>
<tr>
<td>hw_ift()</td>
<td>411</td>
</tr>
<tr>
<td>intr_migr_info_t</td>
<td>331</td>
</tr>
<tr>
<td>iovec</td>
<td>342</td>
</tr>
<tr>
<td>isc_table_type</td>
<td>353</td>
</tr>
<tr>
<td>scb()</td>
<td>539</td>
</tr>
<tr>
<td>scsi_ddsw()</td>
<td>557</td>
</tr>
<tr>
<td>scsi_ifsw()</td>
<td>567</td>
</tr>
<tr>
<td>uio</td>
<td>576</td>
</tr>
<tr>
<td>subsys_N_format()</td>
<td>436</td>
</tr>
<tr>
<td>subsys_N_get_options()</td>
<td>443</td>
</tr>
<tr>
<td>subsys_N_format()</td>
<td>436</td>
</tr>
<tr>
<td>subsys_N_get_options()</td>
<td>443</td>
</tr>
<tr>
<td>support disclaimers</td>
<td>16</td>
</tr>
<tr>
<td>suser()</td>
<td>145</td>
</tr>
<tr>
<td>sw_trigger()</td>
<td>146</td>
</tr>
<tr>
<td>timeout()</td>
<td>149</td>
</tr>
</tbody>
</table>
Index

tl_banner_char() Network function, 447
tl_check_cat_version() Network function, 448
tl_format_fprintf() Network function, 450
tl_format_write() Network function, 453
tl_get_line() Network function, 456
tl_get_parms() Network function, 458
tl_header_format1() Network function, 462
tl_log_class() Network function, 465
tl_rae_format() Network function, 467
tl_trace_kind() Network function, 469
trigger, software, 146

U
uio kernel structure, 151
uiomove() kernel function, 153
unmap_mem_from_host() kernel function, 156
untimout() kernel function, 157
user_ioctl() kernel function, 158
user_iounmap() kernel function, 161

V
VASSERT() kernel function, 162

W
waiting_in_select() kernel function, 163
wakeup() kernel function, 165
WRITE_REG_UINTn_ISC() PCI macros, 495
WSIO device probe structure, 363
WSIO functions
bp_dma_cleanup(), 188
bp_dma_setup(), 190
dma_cleanup(), 192
dma_setup(), 195
driver_addr_probe(), 197
driver_attach(), 199
driver_class_probe(), 204
driver_close(), 201
driver_dev_init(), 203
driver_if_init(), 206
driver_install(), 207
driver_ioctl(), 209
driver_isr(), 211
driver_minor_build(), 212
driver_minphys(), 214
driver_open(), 216
driver_psize(), 218
driver_read(), 219
driver_select(), 221
driver_strategy(), 223
driver_write(), 224
free_isc(), 226
get_new_isc(), 227
iodone(), 229
iowait(), 231
isc_claim(), 233
isrlink(), 600
isrunlink(), 602
m_wsio_funcnum(), 239
m_wsio_selcode(), 240
m_wsio_vsc(), 241
mod_wsio_attach_list_add(), 242
mod_wsio_attach_list_remove(), 244
wsio_activate_probe(), 246
wsio_alloc_mem(), 249
wsio_alloc_mem_handle(), 251
wsio_allocat_shared_memory(), 605
WSIO_BIG_ENDIAN(), 260
wsio_dma_attributes(), 383
wsio_dma_pass_thru(), 266
wsio_dma_set_device_attributes(), 269
wsio_fastmap(), 608
wsio_fastmap_dma_buffer(), 279
wsio_flush_shared_mem(), 282
wsio_flush_shared_memory(), 611
wsio_free_dma_handle(), 284
wsio_free_mem(), 286
wsio_free_mem_handle(), 287
wsio_free_shared_mem(), 288
wsio_free_shared_memory(), 613
wsio_get_active_processor_count(), 292
wsio_get_all_registers(), 293
wsio_get_interrupts(), 615
wsio_get_ioports(), 295
wsio_get_isc(), 297
wsio_get_processor_count(), 299
wsio_get_pva(), 617
wsio_get_registers(), 620
wsio_get_system_params(), 300
wsio_install_driver(), 304
wsio_install_drv_event_handler(), 307
wsio_install_drv_func(), 622
wsio_intr_activate(), 309
wsio_intr_alloc(), 312
wsio_intr_deactivate(), 314
wsio_intr_deactivate_nowait(), 317
wsio_intr_free(), 320
wsio_intr_get_assigned_cpu(), 322
wsio_intr_get_irq_line(), 324
Index

wsio_intr_get_txn_info(), 326
wsio_intr_set_cpu_spec(), 332
wsio_intr_set_irq_line(), 336
wsio_ioa_sync(), 338
wsio_iova_to_phys(), 339
wsio_isc_to_instance(), 341
wsio_map(), 625
wsio_map_cfg_handle(), 345
wsio_map_dma_buffer(), 347
wsio_map_port(), 353
wsio_map_reg(), 355
wsio_port_inXX(), 359
wsio_port_outXX(), 361
wsio_query_supported_function(), 365
wsio_read_regXX(), 367
wsio_reg_drv_capability_mask(), 369
wsio_register_addr_probe(), 371
wsio_register_dev_probe(), 374
wsio_register_probe_func(), 634
wsio_remap(), 636
wsio_remap_dma_buffer(), 377
wsio_set_attributes(), 638
wsio_set_description(), 381
wsio_set_dma_callback(), 388
wsio_unregister_dev_probe(), 402
WSIO macros
m_instance(), 604
wsio_allocate_dma_handle(), 254
wsio_allocate_shared_mem(), 256
wsio_cfg_inXX(), 262
wsio_cfg_outXX(), 264
wsio_init_map_context(), 302
WSIO_LITTLE_ENDIAN(), 343
WSIO_ORDERED_INTERRUPTS(), 357
wsio_unmap_cfg_handle(), 393
WSIO structures
intr_migr_info_t, 331
isc_table_type, 235
wsioDrv_data_t, 274
wsio_drv_info, 276
wsio_event_t, 277
wsio_generic_event_t, 290
wsio_intr_migr_t, 328
wsio_probe_dev_info(), 363
wsio_activate_probe WSIO function, 246
wsio_alloc_mem WSIO function, 249
wsio_alloc_mem_handle WSIO function, 251
wsio_allocat_shared_memory() WSIO function, 605
wsio_allocat_dma_handle() WSIO macro, 254
wsio_allocat_shared_mem() WSIO macro, 256
WSIO_BIG_ENDIAN() WSIO function, 260
wsio_cfg_inXX() WSIO macro, 262
wsio_cfg_outXX() WSIO macro, 264
wsio_dma_pass-thru() WSIO function, 266
wsio_dma_set_device_attributes() WSIO function, 269
wsio_drv_data_t WSIO structure, 274
wsio_drv_info WSIO structure, 276
wsio_event_t WSIO structure, 277
wsio_fastmap() WSIO function, 608
wsio_fastmap_dma_buffer() WSIO function, 279
wsio_flush_shared_mem() WSIO function, 282
wsio_flush_shared_memory() WSIO function, 611
wsio_free_dma_handle() WSIO function, 284
wsio_free_mem WSIO function, 286
wsio_free_mem_handle() WSIO function, 288
wsio_free_shared_mem() WSIO function, 613
wsio_free_shared_memory() WSIO function, 288
wsio_free_shared_memory() WSIO function, 613
wsio_get_active_processor_count() WSIO function, 292
wsio_get_all_registers() WSIO function, 293
wsio_get_controller() WSIO function, 615
wsio_get_isc() WSIO function, 297
wsio_get_processor_count() WSIO function, 299
wsio_get_pva() WSIO function, 617
wsio_get_registers() WSIO function, 620
wsio_get_system_params() WSIO function, 300
wsio_init_map_context() WSIO macro, 302
wsio_install_driver() WSIO function, 304
wsio_install_drv_event_handler() WSIO function, 307
wsio_install_drv_func() WSIO function, 622
wsiointr_activate() WSIO function, 309
wsiointr_alloc() WSIO function, 312
wsio_intr_deactivate() WSIO function, 314
wsio_intr_deactivate_nowait() WSIO function, 317
wsio_intr_free() WSIO function, 320
wsio_intr_get_assigned_cpu() WSIO function, 322
wsio_intr_get_irq_line() WSIO function, 324
wsio_intr_get_txn_info() WSIO function, 326
wsio_intr_migr_t WSIO structure, 328
wsio_intr_set_cpu_spec() WSIO function, 332
wsio_intr_set_irq_line() WSIO function, 336
wsio_io_sync() WSIO function, 338
wsio_iova_to_phys() WSIO function, 339
wsio_isc_to_instance() WSIO function, 341
WSIO_LITTLE_ENDIAN() WSIO macro, 343
wsio_map() WSIO function, 625
wsio_map_cfg_handle() WSIO function, 345
wsio_map_dma_buffer() WSIO function, 347
wsio_map_port() WSIO function, 353
wsio_map_reg() WSIO function, 355
WSIO_ORDERED_INTERRUPTS() WSIO macro, 357
wsio_port_inXX() WSIO function, 359
wsio_port_outXX() WSIO function, 361
wsio_probe_dev_info() WSIO structure, 363
wsio_query_supported_function() WSIO function, 365
wsio_read_regXX() WSIO function, 367
wsio_reg_drv_capability_mask() WSIO function, 369
wsio_register_addr_probe() WSIO function, 371
wsio_register_dev_probe() WSIO function, 374
wsio_register_probe_func() WSIO function, 634
wsio_remap() WSIO function, 636
wsio_remap_dma_buffer() WSIO function, 377
wsio_set_attributes() WSIO function, 638
wsio_set_description() WSIO function, 381
wsio_set_dma_attributes() WSIO function, 383
wsio_set_dma_callback() WSIO function, 388
wsio_uninstall_driver() WSIO function, 390
wsio_uninstall_drv_event_handler() WSIO function, 391
wsio_unmap() WSIO function, 640
wsio_unmap_cfg_handle() WSIO macro, 393
wsio_unmap_dma_buffer() WSIO function, 395
wsio_unmap_port() WSIO function, 398
wsio_unmap_reg() WSIO function, 400
wsio_unregister_dev_probe() WSIO function, 402
wsio_write_regXX() WSIO function, 405