# Table of Contents

## About This Document

## 1 Kernel Reference Pages

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_spinlock(9F)</td>
<td>18</td>
</tr>
<tr>
<td>b_cpsema(9F)</td>
<td>19</td>
</tr>
<tr>
<td>b_initsema(9F)</td>
<td>21</td>
</tr>
<tr>
<td>b_owns_sema(9F)</td>
<td>23</td>
</tr>
<tr>
<td>b_psema(9F)</td>
<td>25</td>
</tr>
<tr>
<td>b_vsema(9F)</td>
<td>26</td>
</tr>
<tr>
<td>bcopy(9F)</td>
<td>28</td>
</tr>
<tr>
<td>biodone(9F)</td>
<td>29</td>
</tr>
<tr>
<td>biowait(9F)</td>
<td>31</td>
</tr>
<tr>
<td>brelse(9F)</td>
<td>32</td>
</tr>
<tr>
<td>buf(9S)</td>
<td>33</td>
</tr>
<tr>
<td>busywait(9F)</td>
<td>37</td>
</tr>
<tr>
<td>bzero(9F)</td>
<td>38</td>
</tr>
<tr>
<td>copyin(9F)</td>
<td>39</td>
</tr>
<tr>
<td>copyout(9F)</td>
<td>40</td>
</tr>
<tr>
<td>cspinlock(9F)</td>
<td>41</td>
</tr>
<tr>
<td>dealloc_spinlock(9F)</td>
<td>43</td>
</tr>
<tr>
<td>delay, delay_sig(9F)</td>
<td>44</td>
</tr>
<tr>
<td>disksort_dequeue(9F)</td>
<td>46</td>
</tr>
<tr>
<td>disksort_enqueue(9F)</td>
<td>47</td>
</tr>
<tr>
<td>disksort_init_queue(9F)</td>
<td>48</td>
</tr>
<tr>
<td>FREE(9F)</td>
<td>49</td>
</tr>
<tr>
<td>get_sleep_lock(9F)</td>
<td>50</td>
</tr>
<tr>
<td>getc(9F)</td>
<td>52</td>
</tr>
<tr>
<td>getcb(9F)</td>
<td>53</td>
</tr>
<tr>
<td>getcf(9F)</td>
<td>54</td>
</tr>
<tr>
<td>getblk(9F)</td>
<td>55</td>
</tr>
<tr>
<td>geterror(9F)</td>
<td>56</td>
</tr>
<tr>
<td>gsignal(9F)</td>
<td>57</td>
</tr>
<tr>
<td>iomap_enter_shared_acc(9F)</td>
<td>58</td>
</tr>
<tr>
<td>iomap_exit_shared_acc(9F)</td>
<td>59</td>
</tr>
<tr>
<td>iomap_pagesize(9F)</td>
<td>60</td>
</tr>
<tr>
<td>iovec(9S)</td>
<td>61</td>
</tr>
<tr>
<td>kdaemon_thread(9F)</td>
<td>62</td>
</tr>
<tr>
<td>kdaemon_thread_create(9F)</td>
<td>64</td>
</tr>
<tr>
<td>kdaemon_thread_exit(9F)</td>
<td>66</td>
</tr>
<tr>
<td>kdaemon_thread_wait(9F)</td>
<td>67</td>
</tr>
<tr>
<td>kernel_iomap(9F)</td>
<td>69</td>
</tr>
<tr>
<td>kernel_iomap_public(9F)</td>
<td>72</td>
</tr>
<tr>
<td>kernel_iounmap(9F)</td>
<td>74</td>
</tr>
<tr>
<td>kfree(9F)</td>
<td>75</td>
</tr>
<tr>
<td>kmallocl(9F)</td>
<td>76</td>
</tr>
<tr>
<td>kmem_arena_alloc(9F)</td>
<td>78</td>
</tr>
<tr>
<td>kmem_arena_attr_init(9F)</td>
<td>80</td>
</tr>
<tr>
<td>kmem_arena_create(9F)</td>
<td>83</td>
</tr>
<tr>
<td>kmem_arena_destroy(9F)</td>
<td>85</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>kr_close_node(9F)</td>
<td>90</td>
</tr>
<tr>
<td>kr_delete_node(9F)</td>
<td>91</td>
</tr>
<tr>
<td>kr_delete_value(9F)</td>
<td>93</td>
</tr>
<tr>
<td>kr_flush(9F)</td>
<td>94</td>
</tr>
<tr>
<td>kr_get_mod_time(9F)</td>
<td>95</td>
</tr>
<tr>
<td>kr_get_node_info(9F)</td>
<td>97</td>
</tr>
<tr>
<td>kr_get_node_names(9F)</td>
<td>99</td>
</tr>
<tr>
<td>kr_get_value(9F)</td>
<td>101</td>
</tr>
<tr>
<td>kr_get_value_names(9F)</td>
<td>103</td>
</tr>
<tr>
<td>kr_get_vinfo(9F)</td>
<td>105</td>
</tr>
<tr>
<td>kr_link_node(9F)</td>
<td>107</td>
</tr>
<tr>
<td>kr_open_node(9F)</td>
<td>109</td>
</tr>
<tr>
<td>kr_release_reference(9F)</td>
<td>111</td>
</tr>
<tr>
<td>kr_set_node_flags(9F)</td>
<td>112</td>
</tr>
<tr>
<td>kr_set_value(9F)</td>
<td>114</td>
</tr>
<tr>
<td>kr_set_value_flags(9F)</td>
<td>116</td>
</tr>
<tr>
<td>Ktimeout(9F)</td>
<td>118</td>
</tr>
<tr>
<td>ktune_canauto(9F)</td>
<td>120</td>
</tr>
<tr>
<td>ktune_current(9F)</td>
<td>121</td>
</tr>
<tr>
<td>ktune_error(9F)</td>
<td>122</td>
</tr>
<tr>
<td>ktune_event_t(9S)</td>
<td>123</td>
</tr>
<tr>
<td>ktune_get(9F)</td>
<td>124</td>
</tr>
<tr>
<td>ktune_handler(9F)</td>
<td>125</td>
</tr>
<tr>
<td>ktune_id(9F)</td>
<td>128</td>
</tr>
<tr>
<td>ktune_inactive(9F)</td>
<td>129</td>
</tr>
<tr>
<td>ktune_isauto(9F)</td>
<td>130</td>
</tr>
<tr>
<td>ktune_isdefault(9F)</td>
<td>131</td>
</tr>
<tr>
<td>ktune_isdynamic(9F)</td>
<td>132</td>
</tr>
<tr>
<td>ktune_name(9F)</td>
<td>133</td>
</tr>
<tr>
<td>ktune_pending(9F)</td>
<td>134</td>
</tr>
<tr>
<td>ktune_register_handler(9F)</td>
<td>136</td>
</tr>
<tr>
<td>ktune_savedefault(9F)</td>
<td>138</td>
</tr>
<tr>
<td>ktune_simple_constraint(9F)</td>
<td>139</td>
</tr>
<tr>
<td>ktune_simple_constraint_data_t(9S)</td>
<td>142</td>
</tr>
<tr>
<td>ktune_simple_dynamic(9F)</td>
<td>143</td>
</tr>
<tr>
<td>ktune_unregister_handler(9F)</td>
<td>145</td>
</tr>
<tr>
<td>ktune_validate_powerof2(9F)</td>
<td>147</td>
</tr>
<tr>
<td>ktune_validate_zero_or_min(9F)</td>
<td>149</td>
</tr>
<tr>
<td>ktune_warning(9F)</td>
<td>151</td>
</tr>
<tr>
<td>ldSID(9F)</td>
<td>152</td>
</tr>
<tr>
<td>major(9F)</td>
<td>153</td>
</tr>
<tr>
<td>makedev(9F)</td>
<td>154</td>
</tr>
<tr>
<td>MALLOCC(9F)</td>
<td>155</td>
</tr>
<tr>
<td>map_mem_to_host(9F)</td>
<td>157</td>
</tr>
<tr>
<td>minor(9F)</td>
<td>158</td>
</tr>
<tr>
<td>minphys(9F)</td>
<td>159</td>
</tr>
<tr>
<td>msg_printf(9F)</td>
<td>160</td>
</tr>
<tr>
<td>owns_spinlock(9F)</td>
<td>161</td>
</tr>
<tr>
<td>p_pgrp(9F)</td>
<td>163</td>
</tr>
<tr>
<td>panic(9F)</td>
<td>164</td>
</tr>
<tr>
<td>physio(9F)</td>
<td>165</td>
</tr>
<tr>
<td>printf(9F)</td>
<td>167</td>
</tr>
<tr>
<td>privileged_cred(9F)</td>
<td>168</td>
</tr>
</tbody>
</table>
driver_strategy(9E)......................................................... 243
driver_write(9E)............................................................ 244
drv_info(9S)................................................................. 245
drv_ops(9S)................................................................. 247
free_isc(9F)................................................................. 249
get_new_isc(9F)............................................................ 250
install_driver(9F)......................................................... 251
io_hw_path_to_node(9F)................................................. 252
io_hw_path_to_str(9F)................................................. 253
io_mkdev(9F)............................................................... 254
io_node_to_hw_path(9F)................................................. 255
io_query(9F)............................................................... 256
io_search(9F)............................................................. 257
io_str_to_hw_path(9F)................................................... 262
iodone(9F)................................................................. 263
iowait(9F)................................................................. 264
isc_claim(9F).............................................................. 265
isc_table_type(9S)......................................................... 267
m_wsio_funcnum(9F)..................................................... 270
m_wsio_selcode(9F)....................................................... 271
m_wsio_vsc(9F)........................................................... 272
mod_wsio_attach_list_add(9F)......................................... 273
mod_wsio_attach_list_remove(9F)................................. 274
modlink(9S)............................................................... 275
modwrapper(9S).......................................................... 276
wsio_activate_probe(9F)................................................. 277
wsio_alloc_mem(9F)....................................................... 279
wsio_alloc_mem_handle(9F).......................................... 280
wsio_allocate_dma_handle(9F)....................................... 282
wsio_allocate_shared_mem(9F)....................................... 283
WSIO_BIG_ENDIAN(9F)................................................... 286
wsio_cfg_inXX(9F)......................................................... 287
wsio_cfg_outXX(9F)....................................................... 288
wsio_create_attribute(9F)............................................. 289
wsio_create_interface(9F)............................................. 291
wsio_destroy_attribute(9F).......................................... 293
wsio_destroy_interface(9F).......................................... 294
wsio_dma_pass_thru(9F)................................................ 295
wsio_dma_set_device_attributes(9F).............................. 297
wsio_drv_data(9S)....................................................... 300
wsio_drv_info(9S)....................................................... 301
wsio_fastmap_dma_buffer(9F)....................................... 302
wsio_flush_shared_mem(9F).......................................... 304
wsio_free_dma_handle(9F)............................................ 306
wsio_free_mem(9F)...................................................... 307
wsio_free_mem_handle(9F)........................................... 308
wsio_free_shared_mem(9F)............................................ 309
wsio_get_active_processor_count(9F)............................ 311
wsio_get_all_registers(9F).......................................... 312
wsio_get_attribute(9F)............................................... 313
wsio_get_iports(9F).................................................... 314
wsio_get_isc(9F)....................................................... 316
wsio_get_processor_count(9F)....................................... 317
wsio_get_relationship(9F).......................................... 318
wsio_get_system_params(9F)......................................... 319
3 PCI Reference Pages.............................................................................................................417
CONNECT_INIT_ROUTINE(9F).......................................................................................................418
PCI_ATTACH_DEV_INIT_ERROR(9F).................................................................................................419
pci_desc_bus_transactions_isc(9F)................................................................................................420
pci_get_port_hdl_isc(9F).............................................................................................................422
pci_read_cfg_uintN_isc(9F).........................................................................................................424
pci_read_port_uintN_isc(9F)........................................................................................................426
pci_unget_port_hdl_isc(9F).........................................................................................................428
pci_write_cfg_uintN_isc(9F)........................................................................................................429
pci_write_port_uintN_isc(9F)......................................................................................................430
READ_REG_UINTn_ISC(9F).............................................................................................................431
WRITE_REG_UINTn_ISC(9F)............................................................................................................433
PCI_ERRATA-1(9).......................................................................................................................435
PCI_ERRATA-2(9).......................................................................................................................437
PCI_ERRATA-3(9).......................................................................................................................438

4 Network Device Driver Reference Pages.................................................................................439
DL_DISABMULTI_REQ(9G)............................................................................................................440
DL_ENABMULTI_REQ(9G)..............................................................................................................441
DL_GET_STATISTICS_REQ(9G).....................................................................................................442
dl_hp_autoneg_sense_t(9S).........................................................................................................443
dl_hp_create_info_t(9S)..............................................................................................................444
dl_hp_dest_llc_info_t(9S)...........................................................................................................449
dl_hp_drv_features_one_t(9S)....................................................................................................450
dl_hp_drv_param_req_type_t(9S)................................................................................................453
dl_hp_duplex_mode_t(9S)............................................................................................................455
dl_hp_encaps_type_t(9S)............................................................................................................456
dl_hp_event_link_cause_t(9S)....................................................................................................457
dl_hp_event_type_t(9S)................................................................................................................459
DL_HP_GET_DRV_PARAM_IOCTL(9G)..............................................................................................461
DL_HP_GET_MIBSTATS_REQ(9G)....................................................................................................463
DL_HP_GET_64BIT_STATS_REQ(9G)..............................................................................................464
dl_hp_getinfo_t(9S).......................................................................................................................465
dl_hp_hdr_length_t(9S)..............................................................................................................467
DL_HP_HW_RESET_REQ(9G).........................................................................................................468
dl_hp_hw_state_t(9S)....................................................................................................................469
dl_hp_ifadmin_state_t(9S).........................................................................................................470
dl_hp_instance_info_t(9S)..........................................................................................................471
dl_hp_llc_info_t(9S)....................................................................................................................474
dl_hp_mac_type_t(9FS)..................................................................................................................475
dl_hp_oop_hdr_t(9S)....................................................................................................................476
dl_hp_oop_template_t(9S)............................................................................................................477
dl_hp_oop_type_t(9S)....................................................................................................................478
dl_hp_op_param_type_t(9S).........................................................................................................479
dl_hp_op_t(9S)............................................................................................................................480
dl_hp_pkt_type_t(9S)....................................................................................................................481
dl_hp_promise_state_t(9S)..........................................................................................................484
dl_hp_prop_t(9S)..........................................................................................................................485
DL_HP_RESET_STATS_REQ(9G)......................................................................................................493
dl_hp_search_info_t(9S)..............................................................................................................494
5 SCSI Reference Pages

dd_close(9E) ................................................................. 566
dd_done(9E) ............................................................... 567
dd_ioctl(9E) ............................................................... 568
dd_ioctl_okay(9E) ...................................................... 569
dd_open(9E) ............................................................... 570
dd_pass_thru_done(9E) .............................................. 571
dd_pass_thru_okay(9E) .............................................. 572
A Deprecated and Obsoleted Interfaces..............................................................................................629
List of Tables

1   HP-UX 11i Releases.......................................................................................................................14
1-1 Relevant buf Structure...................................................................................................................33
1-2 iovec Structure Fields....................................................................................................................61
1-3 uio Structure Fields......................................................................................................................197
2-1 Device Driver Fields....................................................................................................................247
2-2 Driver Relevant Structure Fields....................................................................................................267
2-3 Driver-Initialized ISC Fields........................................................................................................268
2-4 Modlink Structure.........................................................................................................................275
2-5 Modwrapper Structure....................................................................................................................276
2-6 Values for type................................................................................................................................363
2-7 Values for assign_type....................................................................................................................365
2-8 Values for capability......................................................................................................................368
2-9 Values for hints.............................................................................................................................375
2-10 Values for type............................................................................................................................377
4-1 Structure Fields............................................................................................................................526
4-2 Structure Fields (ss_N_fmt_flag_type)........................................................................................542
4-3 Structure Fields (ss_N_fmt_parms_type)......................................................................................544
5-1 Relevant scb Structure Fields.......................................................................................................593
5-2 Relevant scsi_ifsw Structure Fields..............................................................................................611
A-1 Deprecated Interfaces, Their Description, and Replacement......................................................629
About This Document

This manual provides kernel, WSIO, GIO, PCI, networking, and SCSI reference manpages for HP-UX 11i v2 platforms.

You can find the latest version of this document on the web at:

http://www.hp.com/go/hpux_ddk

Intended Audience

This document is intended for system administrators or developers responsible for porting or writing drivers. Developers are expected to have:

- Experience writing programs in the C language.
- Working knowledge of the basic concepts of writing a driver.
- An understanding of the functionality of the hardware for which the driver is being written.
- Read the HP-UX System Administration Tasks manual and performed system administration.
- Working knowledge of the virtual memory, I/O, and file system areas in the HP-UX and/or UNIX operating systems.

This document is not a tutorial; it is intended for reference.

What’s in This Document

The HP-UX 11i v2 Driver Development Reference (DDR) is divided into several chapters, and each contains manpage information:

- **Chapter 1 (page 17)** contains manpages for the kernel support routines commonly used by I/O drivers.
- **Chapter 2 (page 211)** contains manpages describing routines and data structures used by drivers to communicate with the WSIO CDIO. It also includes manpages for GIO routines and data structures.
- **Chapter 3 (page 417)** describes driver functions that are specific to PCI Services.
- **Chapter 4 (page 439)** contains manual reference pages for data structures, kernel and user space support routines, and macros essential for HP-UX network interface drivers.
- **Chapter 5 (page 565)** contains SCSI reference pages and a set of commonly used SCSI functions that allow device and interface drivers to be smaller and more supportable.
- **Appendix A (page 629)** contains a list of manpages that have been deprecated or obsoleted in HP-UX 11i v2.

Typographical Conventions

This document uses the following conventions.

- **audit(5)**: An HP-UX manpage. In this example, audit is the name and 5 is the section in the HP-UX Reference. On the web and on the Instant Information CD, it may be a hot link to the manpage itself. From the HP-UX command line, you can enter “man audit” or “man 5 audit” to view the manpage. See man(1).

- **Book Title**: The title of a book. On the web and on the Instant Information CD, it may be a hot link to the book itself.

- **KeyCap**: The name of a keyboard key. Note that **Return** and **Enter** both refer to the same key.

- **Emphasis**: Text that is emphasized.

- **Bold**: Text that is strongly emphasized.

- **Bold**: The defined use of an important word or phrase.
ComputerOut Text displayed by the computer.
UserInput Commands and other text that you type.
Command A command name or qualified command phrase.
Variable The name of a variable that you may replace in a command or function or information in a display that represents several possible values.
[] The contents are optional in formats and command descriptions. If the contents are a list separated by |, you must choose one of the items.
{} The contents are required in formats and command descriptions. If the contents are a list separated by |, you must choose one of the items.
... The preceding element may be repeated an arbitrary number of times.
| Separates items in a list of choices.

HP-UX Release Name and Release Identifier

Each HP-UX 11i release has an associated release name and release identifier. The `uname` command with the `-r` option returns the release identifier. This table shows the releases available for HP-UX 11i.

**Table 1 HP-UX 11i Releases**

<table>
<thead>
<tr>
<th>Release Identifier</th>
<th>Release Name</th>
<th>Supported Processor Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.11.11</td>
<td>HP-UX 11i v1</td>
<td>PA-RISC</td>
</tr>
<tr>
<td>B.11.20</td>
<td>HP-UX 11i v1.5</td>
<td>Intel® Itanium®</td>
</tr>
<tr>
<td>B.11.22</td>
<td>HP-UX 11i v1.6</td>
<td>Intel® Itanium®</td>
</tr>
<tr>
<td>B.11.23</td>
<td>HP-UX 11i v2.0</td>
<td>Intel® Itanium®</td>
</tr>
<tr>
<td>B.11.23</td>
<td>HP-UX 11i v2.0 September 2004</td>
<td>PA RISC and Intel® Itanium®</td>
</tr>
<tr>
<td>B.11.23</td>
<td>HP-UX 11i v2.0 June 2007</td>
<td>PA RISC and Intel® Itanium®</td>
</tr>
</tbody>
</table>

Related Documents

Additional information about the DDR can be found at:

http://www.hp.com/go/hpux_ddk

Other documents in the DDK collection include:

- **DDK FAQ**
- **HP-UX 11i v2 Driver Development Kit Getting Started Guide**
- **HP-UX 11i v2 Driver Development Guide**

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs.

Send comments to:

feedback@fc.hp.com

Include the document title and manufacturing part number with your comments.
Email and Internet Resources

HPHP program and developer resource materials are available at the following locations:

- HPHP Program E-mail at: hphp.support@hp.com
- HP Hardware Provider program at: http://www.hp.com/go/hphp

Support and Compatibility Disclaimers

Because drivers function at the level of the kernel, HP reminds you of the following:

- Adding your own driver to HP-UX requires relinking the driver into HP-UX. With each new release you must 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 (you can have some system dependencies).
- 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.
- If difficulties arise during the development and test phases of writing a driver, HP might 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 must seek help from the third party from whom you obtained software or hardware.
This chapter contains reference pages for the kernel support routines commonly used by kernel modules.
alloc_spinlock(9F)

NAME

alloc_spinlock - Allocate and initialize a spinlock resource.

SYNOPSIS

#include <sys/spinlock.h>

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

PARAMETERS

order The lock order of the spinlock.
name Pointer to a character string containing the name of the spinlock.

DESCRIPTION

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(9F).

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>.

RETURN VALUES

The alloc_spinlock returns a pointer to a spinlock resource.

CONSTRAINTS

Do not call in an interrupt context.
Do not call 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(9F), dealloc_spinlock(9F), spinunlock(9F), spinlock(9F)
NAME

b_cpsema - 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

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.

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 (sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

RETURN VALUES

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

CONSTRAINTS

Do not call in an interrupt context.

Do not call 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_cpsema(&mydrv_sema_2);
...

*/

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

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

/*
 * Now acquire the two beta semaphores in the correct
 * order to avoid a deadlock. We will probably block
 * here until the other thread releases mydrv_sema_1.
 */
 b_psema(&mydrv_sema_1);
 b_psema(&mydrv_sema_2);

/*
 * Check if data have changed between the time we released
 * mydrv_sema_2 and reacquired the beta semaphores.
 */
 

SEE ALSO

b_initsema(9F), b_owns_sema(9F), b_psema(9F), b_vsema(9F)
**b_initsema(9F)**

**NAME**

b_initsema - Initialize a beta semaphore.

**SYNOPSIS**

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

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

**PARAMETERS**

- **sema** Pointer to a b_sema structure.
- **val** Initial value of `sema`. Normally set to 1.
- **order** Lock order.
- **name** Beta semaphore name.

**DESCRIPTION**

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

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

The `order` parameter is the lock order of the beta semaphore. It must be a positive value and chosen so that deadlocks with other beta semaphores are avoided. To avoid deadlocks, the semaphore to be acquired must have a lock order greater than that of any semaphore currently held by the thread. See beta-class semaphore lock orders in `<sys/semglobal.h>`.

If the `SEMA_DEADLOCK_SAFE` flag is set (ORed with `order`), deadlock detection is disabled for the special case where beta semaphores have the same lock order values. The caller ensures 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.

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 (sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

**RETURN VALUES**

None

**CONSTRAINTS**

Do not call in an interrupt context.

Do not call 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(9F), b_owns_sema(9F), b_psema(9F), b_vsema(9F)
b_owns_sema(9F)

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

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 (sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

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
Do not call in an interrupt context.

EXAMPLE
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*(9F), *b_initsema*(9F), *b_psema*(9F), *b_vsema*(9F)
b_psema(9F)

NAME

b_psema - Acquire (lock) a beta semaphore.

SYNOPSIS

#include <sys/sem_beta.h>

void b_psema(
    struct b_sema *sema
);

PARAMETERS

sema Pointer to a b_sema structure.

DESCRIPTION

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

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 (sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

RETURN VALUES

None

CONSTRAINTS

Do not call in an interrupt context.
Do not call while holding a spinlock.

EXAMPLE

static b_sema_t mydrv_sema;
...

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

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

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

SEE ALSO

b_cpsema(9F), b_initsema(9F), b_owns_sema(9F) b_vsema(9F)
NAME
b_vsema - 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
The b_vsema kernel function releases (unlocks) the beta semaphore pointed to by sema.
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 (sleep) and still keep ownership. Threads waiting to acquire an owned beta
semaphore are blocked.

RETURN VALUES
None

CONSTRAINTS
Do not call in an interrupt context.

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

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

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

SEE ALSO
b_cpsema(9F), b_psema(9F), b_owns_sema(9F), b_initsema(9F)
NAME

bcmp - 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, the function returns zero. If the \( n \) bytes are not identical, the function returns the integer value of \((s1[k] - s2[k])\), 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

None

SEE ALSO

strcmp(9F)

strcmp(9F)
NAME
bcopy - Copy data from a source buffer to a destination buffer

SYNOPSIS
#include <sys/kern_svcs.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 uiomove. To copy data between buffers in other user spaces, use privlbcopy.

RETURN VALUES
None

CONSTRAINTS
None

WARNINGS
Do not use the bcopy function for transfers between memory and I/O space. The underlying routines make choices of the optimal transfer code that 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
copyin(9F), copyout(9F), privlbcopy(9F), uiomove(9F)
NAME

biodone - 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 must 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

Do not call 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(9F), buf(9F)
NAME

biowait - 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

Do not call in an interrupt context.
Do not call while holding a spinlock.

EXAMPLES

int error;
struct buf *bp;
...

/*
 * After starting the I/O request, wait for its completion.
 */
error = biowait(bp);

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

SEE ALSO

biodone(9F), buf(9F), geterror(9F)
NAME
brelse – 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 must 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
Do not call while holding a spinlock of order >= BUF_FREE_LOCK_ORDER.

SEE ALSO
buf(9F), geteblk(9F)
buf - File system buffer header structure for block I/O.

#include <sys/buf.h>

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>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>av_back</td>
<td>struct buf *</td>
</tr>
<tr>
<td>av_forw</td>
<td>struct buf *</td>
</tr>
<tr>
<td>b_bcount</td>
<td>int32_t</td>
</tr>
<tr>
<td>b_blkno</td>
<td>daddr_t</td>
</tr>
<tr>
<td>b_bufsize</td>
<td>int32_t</td>
</tr>
<tr>
<td>b_dev</td>
<td>dev_t</td>
</tr>
<tr>
<td>b_error</td>
<td>short</td>
</tr>
<tr>
<td>b_flags</td>
<td>int32_t</td>
</tr>
<tr>
<td>b_iiodone</td>
<td>int (*)(())</td>
</tr>
<tr>
<td>b_merge</td>
<td>struct buf *</td>
</tr>
<tr>
<td>b_merge_cnt</td>
<td>uint16_t</td>
</tr>
<tr>
<td>b_resid</td>
<td>unsigned int</td>
</tr>
<tr>
<td>b_s2</td>
<td>intptr_t</td>
</tr>
<tr>
<td>b_s3</td>
<td>char</td>
</tr>
<tr>
<td>b_s7</td>
<td>intptr_t</td>
</tr>
<tr>
<td>b_s8</td>
<td>intptr_t</td>
</tr>
<tr>
<td>b_spaddr</td>
<td>space_t</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>b_un.b_addr</td>
<td>caddr_t</td>
</tr>
<tr>
<td>b2_flags</td>
<td>uint16_t</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:
  - **Flag** | **Meaning if Set**
  - `B_ASYNC` | Buffer write is synchronous. Do not wait for I/O completion. Mutually exclusive with `B_SYNC`.
  - `B_BCACHE` | The buffer is allocated from the file system buffer cache.
  - `B_BUSY` | The buffer is in use.
  - `B_CACHE` | `bread` located this buffer in the cache.
  - `B_CALL` | `iodone` is to call the function pointed to by `b_iiodone`.
  - `B_DELWRI` | Delayed write. Write at exit of avail list processing by the buffer cache management code.
  - `B_DONE` | The buffer transfer has completed; `biodone` sets this flag.
  - `B_END_OF_DATA` | This flag is used to terminate, without error, a `physio` transfer, with less than `b_count` bytes transferred.
  - `B_ERROR` | An error occurred during the I/O transfer. If the driver sets this flag, it must also set the `b_error` field with an `errno` value.
  - `B_FSYSIO` | Buffer came from `bread` or `bwrite`.
  - `B_INVAL` | The buffer does not contain valid information.
  - `B_NDELAY` | Do not retry on failures.
  - `B_NOCACHE` | Do not cache data buffer when released.
  - `B_PAGEOUT` | This flag is used by the buffer cache management system and must 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 must not be touched by a driver.

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

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

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

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

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

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

b_iiodone 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.

b_merge 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.

b_merge_cnt Number of buffers merged together through b_merge.

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

b_s2 Scratch field for driver use. For example, SCSI Interface Drivers use this field to store a pointer to a SCSI Control Block.

b_s3 Scratch field for driver use. For example, the SCSI Subsystem uses this field to store state information.

b_s7 Scratch field for driver use. For example, the SCSI Subsystem reserves this field for device drivers.

b_s8 Scratch field for driver use. For example, the SCSI subsystem reserves this field for device drivers.

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

b_un.b_addr Virtual address of the buffer specified by the buffer header. The buffer may be mapped in kernel space or it may be mapped in user space. If mapped in user space, b_spaddr must be used with b_addr to form the global virtual address in order to access the buffer. See privlbcopy(9F) for more information.
$b2\_flags$  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>Meaning if Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2_LIST</td>
<td>Buffer is linked with other buffers through the $b_merge$ field.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

$biodone$(9F), $biovait$(9F), $brelse$(9F), $geteblk$(9F), $geterror$(9F), $physio$(9F), $privlbcopy$(9F)
NAME
busywait - Wait at least \( t \) microseconds by spinning processor cycles.

SYNOPSIS
#include <sys/kern_svcs.h>
void busywait(
    ulong_t \( t \)
);

PARAMETERS
\( t \)  The wait time in microseconds.

DESCRIPTION
The \texttt{busywait} kernel function waits at least \( t \) microseconds by spinning processor cycles. The processor does no useful work while busy waiting, so keep \( t \) as short as possible.

The \texttt{busywait} does not block (that is, sleep), and can be called in an interrupt context or while holding a spinlock. If \texttt{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.

Use the \texttt{delay} kernel function instead if the desired behavior is to block and relinquish the processor.

RETURN VALUES
None

CONSTRAINTS
None

EXAMPLES
/*
 * busywait for 5 microseconds
 */

busywait(5UL);

WARNINGS
On a uniprocessor, it is possible for \texttt{busywait} to hang the system since it does not yield the processor if it is called in interrupt context.

SEE ALSO
delay(9F), timeout(9F), untimout(9F).
NAME
bzero – 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
None

SEE ALSO
bcopy(9F)
copyin(9F)

NAME

copyin - 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.

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.
The copyin function might block (sleep); do not call it while holding a spinlock.

RETURN VALUES

0       Successful completion.
<>0     Error.

CONSTRAINTS

Do not call in an interrupt context.
Do not call 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 (for
example, by calling kmalloc). When large buffers are allocated on the kernel stack, the kernel
stack may overflow and cause the kernel to panic.

EXAMPLE

char my_buff[128];
/*
 * Copy from the user buffer to my_buff[] on the kernel stack.
 * Note that buffers larger than 128 bytes must be
 * allocated from kernel memory by calling kmalloc().
 */

if (copyin(user_buf, my_buf, sizeof(my_buf))) {
    return EFAULT;
}

SEE ALSO

bcopy(9F), copyout(9F), privlbcopy(9F), uimoremove(9F)
NAME

copyout – Copy data from a kernel buffer to a user buffer.

SYNOPSIS

#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 \( \text{from} \_\text{kernel} \) to the user space address \( \text{to} \_\text{user} \). 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.

The copyout might block (sleep); do not call it while holding a spinlock.

RETURN VALUES

0 Successful completion.
<>0 Error.

CONSTRAINTS

Do not call in an interrupt context.
Do not call 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 (for example, by calling kmalloc). When large buffers are allocated on the kernel stack, the kernel stack may overflow and cause the kernel to panic.

EXAMPLE

char my_buff[128];

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

SEE ALSO

bcopy(9F), copyin(9F), privlbcopy(9F), uiomove(9F)
NAME
cspinlock -- Conditionally acquire (lock) a spinlock.

SYNOPSIS
#include <sys/spinlock.h>

int cspinlock(
    lock_t *lock
);

PARAMETERS
lock Pointer to a lock_t structure.

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

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.

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

CONSTRAINTS
None

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(9F), dealloc_spinlock(9F), spinlock(9F), spinunlock(9F)
dealloc_spinlock(9F)

NAME

dealloc_spinlock - Deallocate a spinlock resource.

SYNOPSIS

#include <sys/spinlock.h>

void dealloc_spinlock(
    lock_t *lock
);

PARAMETERS

lock Pointer to a lock_t structure.

DESCRIPTION

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

RETURN VALUES

None

CONSTRAINTS

None

SEE ALSO

alloc_spinlock(9F), cspinlock(9F), get_sleep_lock(9F), owns_spinlock(9F), spinlock(9F), spinunlock(9F)
**delay, delay_sig(9F)**

**NAME**
delay, delay_sig - Suspend the calling thread for a specified number of clock ticks.

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

void delay(
    clock_t ticks
);

int delay_sig(
    clock_t ticks
);

**PARAMETERS**
ticks Specifies time value in number of clock ticks. If ticks is zero, the function returns immediately.

**DESCRIPTION**
The delay kernel function suspends the calling thread for ticks number of clock ticks. The thread sleeps in a non-interruptible mode during this time, and cannot be interrupted by signals until the specified time has expired.

The delay_sig kernel function also suspends the calling thread for ticks number of clock ticks, but it sleeps interruptibly. The delay_sig function indicates through a return value if it was interrupted by a signal.

The Hz value defined in <sys/param.h> file defines the number of clock ticks per second. Although ticks is defined to be type clock_t, the constant MAXINT defined in <sys/param.h> specifies the actual limit on the number of ticks that a thread can sleep. If ticks is greater than this maximum, it is silently rounded down to it.

The delay and delay_sig functions do not busy wait. Use the busywait function if busy waiting is needed.

Use the timeout function to asynchronously execute some function after a delay of specified number of clock ticks.

**RETURN VALUES**
delay Has no return value.
delay_sig Returns the following values:
  0   The sleep completed normally and was not interrupted.
  EINTR The sleep was interrupted by a signal.

**CONSTRAINTS**
Do not call while holding any spinlocks.
Do not call from interrupt context.

**EXAMPLES**
/*
 * Block this thread for three seconds, and then resume.
 */

delay(3*HZ);
SEE ALSO

`busywait(9F), timeout(9F), untimeout(9F)`.
NAME
disksort_dequeue - Dequeue a buffer from the disksort queue.

SYNOPSIS
#include <sys/disksort.h>
struct buf*disksort_dequeue(
    struct buf *dp
);

PARAMETERS
dp  Pointer to the head of the disksort queue.

DESCRIPTION
The disksort_dequeue function dequeues a buffer from the disksort queue pointed to by dp. The enqueued buffers are sorted in an order that attempts to minimize seek and rotational latencies experienced by the disk device. The order in which buffers are dequeued may differ from the order in which the driver enqueued the buffers.

The disk driver is responsible for synchronizing access to the disksort queue. The disk driver must use a lock, such as a spinlock, when calling disksort_enqueue and disksort_dequeue.

RETURN VALUES
<>NULL  Pointer to the buffer dequeued.
NULL    No buffer dequeued.

CONSTRAINTS
May be called in user context or interrupt context.
May be called while holding a spinlock.

EXAMPLE
lock_t *mylock = abc;  /* Pointer to driver spinlock */
struct buf *dp = xyz;  /* Pointer to head of disksort queue */
struct buf *bp;  /* Pointer to buffer dequeued */
...
spinlock(mylock);
bp = disksort_dequeue(dp);
spinunlock(mylock);
if (bp != NULL)
{
    /* Issue the buffer dequeued to the disk device. */
}

SEE ALSO
disksort_enqueue(9F), disksort_init_queue(9F)
NAME

disksort_enqueue - Enqueue a buffer onto the disksort queue.

SYNOPSIS

#include <sys/disksort.h>

void disksort_enqueue(
    struct buf *dp,
    struct buf *bp
);

PARAMETERS

dp  Pointer to the head of the disksort queue.
bp  Pointer to the buffer to be enqueued.

DESCRIPTION

The disksort_enqueue function enqueues the buffer onto the disksort queue pointed to
by dp. The enqueued buffers are sorted in an order that attempts to minimize seek and rotational
latencies experienced by the disk device. The order in which buffers are dequeued might differ
from the order in which the driver enqueued the buffers.

The disk driver is responsible for synchronizing access to the disksort queue. The disk driver
must hold a lock, such as a spinlock, when calling disksort_enqueue and disksort_dequeue.

CONSTRAINTS

May be called in user context or interrupt context.
May be called while holding a spinlock.

EXAMPLE

mydriver_strategy(struct buf *bp)
{
    lock_t *mylock = abc;    /* Pointer to driver spinlock */
    struct buf *dp = xyz;    /* Pointer to head of disksort queue */
    ...
    spinlock(mylock);
    disksort_enqueue(dp, bp);
    spinunlock(mylock);
}

SEE ALSO

disksort_dequeue(9F), disksort_init_queue(9F)
NAME

disksort_init_queue - Initialize the disksort queue.

SYNOPSIS

#include <sys/disksort.h>

void disksort_init_queue(
    struct buf *dp
);

PARAMETERS

dp  Pointer to the head of the disksort queue.

DESCRIPTION

The disksort_init_queue function initializes the buf structure defining the disksort queue. The queue contains a sorted list of buffers that have been enqueued by a disk driver instance. The disk driver must allocate storage for the buf structure, pointed to by dp, that is used as the head of the disksort queue.

CONSTRAINTS

May be called in user context or interrupt context.
May be called while holding a spinlock.

EXAMPLE

static int
mydriver_open(dev_t dev, int oflags)
{
    struct buf *dp = xyz;  /* Pointer to head of disksort queue */
    ... 
    if (is_first_open) {
        disksort_init_queue(dp);
    }
    ... 
}

SEE ALSO

disksort_dequeue(9F), disksort_enqueue(9F)
FREE(9F)

NAME
FREE - 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.

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
None

EXAMPLES
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(9F), kfree(9F), MALLOC(9F)
get_sleep_lock(9F)

NAME

get_sleep_lock -- Acquire a sleep queue spinlock.

SYNOPSIS

#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.

The get_sleep_lock function 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.

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

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

NOTE: The activity_count variable is protected by a sleep queue spinlock, which is released by sleep. When the asynchronous activity completes, the corresponding completion routine calls get_sleep_lock before it decrements activity_count and calls wakeup.

    spinunlock(sleep_lock);

The wakeup function 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 preceding manner.

RETURN VALUES

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

CONSTRAINTS

Do not call 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(9F), wakeup(9F)
NAME

getc - 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

None

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);
}

SEE ALSO

putc(9F), getcb(9F), getcf(9F), putcb(9F), putcf(9F)
NAME
getcb - Manipulate cblocks on a clist.

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

PARAMETERS

list Pointer to a clist.

DESCRIPTION
The 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
None

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(9F), getc(9F), getcf(9F), putc(9F), putcf(9F)
NAME
getcf - Get a cblock on the cblock free list.

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

PARAMETERS
None

DESCRIPTION
The getcf function gets the next free cblock on the cfreelist.

RETURN VALUES
<>NULL  Successful completion. The value is a pointer to the cblock.
NULL    Error.

CONSTRAINTS
None

SEE ALSO
getc(9F), getcb(9F)
geteblk(9F)

NAME
geteblk - 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 must be used for short durations and released by calling brelse when no longer needed.

RETURN VALUES
The geteblk returns a pointer to the allocated buffer header.

CONSTRAINTS
Do not call in an interrupt context.
Do not call while holding a spinlock.

EXAMPLE
struct buf * bp;

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

SEE ALSO
brese(9F), buf(9F)
geterror(9F)

NAME

gterror - Return the error number from the buffer header.

SYNOPSIS

#include <sys/buf.h>

int geterror(
    struct buf *bp
);

PARAMETERS

bp Pointer to a buf structure.

DESCRIPTION

The geterror 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

None

EXAMPLE

int err;
struct buf *bp;
...

/*
 * After starting the I/O request, wait for its completion.
 */
err = 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 (err) {
    err = geterror(bp);
}

SEE ALSO

biowait(9F), buf(9F)
NAME

gsignal - Send the specified signal to all processes in a process group.

SYNOPSIS

#include <sys/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 p_grp kernel function.
Signal numbers are defined in <sys/signal.h>.

RETURN VALUES

None

CONSTRAINTS

This interface cannot be called from Interrupt Control Stack (ICS).

EXAMPLE

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

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

SEE ALSO

p_pgrp(9F), psignal(9F)
iomap_enter_shared_acc(9F)

NAME
iomap_enter_shared_acc - Enter code that accesses shared I/O memory.

SYNOPSIS
#include <sys/iomap.h>

iomap_acc_t iomap_enter_shared_acc(void);

PARAMETERS
None

DESCRIPTION
The iomap_enter_shared_acc function returns an opaque handle that allows the kernel to access memory mapped I/O pages that are shared with a user application. The user application is granted access through the user_iomap function and the kernel is granted access through the kernel_iomap function.

Kernel modules must call iomap_enter_shared_acc before accessing a memory mapped I/O page that is shared with a user process, otherwise, the kernel may take a protection trap and panic. When done accessing the memory mapped I/O page, the kernel module must call iomap_exit_shared_acc.

RETURN VALUES
iomap_acc_t opaque handle.

CONSTRAINTS
Can be called in user or interrupt context.
Can be called while holding spinlocks.

EXAMPLES
iomap_acc_t iomap_acc;

iomap_acc = iomap_enter_shared_acc();

/*
 * Read an I/O register that is shared with and accessible by a user application.
 */
ioreg = *iop;

iomap_exit_shared_acc(iomap_acc);

SEE ALSO
iomap_exit_shared_acc(9F), kernel_iomap(9F), kernel_iounmap(9F), user_iomap(9F), user_iounmap(9F).
NAME

iomap_exit_shared_acc - Exit code that accesses shared I/O memory.

SYNOPSIS

#include <sys/iomap.h>

void iomap_exit_shared_acc(
    iomap_acc_t iomap_acc
);

PARAMETERS

iomap_acc Opaque handle returned by iomap_enter_shared_acc.

DESCRIPTION

The iomap_enter_shared_acc function is called in order for a kernel module to access memory mapped I/O locations that it shares with a user space application. The kernel module can share access by first mapping a range of I/O locations to the user through user_iomap. The kernel module then maps the same I/O locations to a kernel space address by calling kernel_iomap passing PROT_URW.

Before the kernel module can access the shared memory mapped I/O locations, it must call iomap_enter_shared_acc, otherwise, a protection key violation fault may be taken. When done accessing the I/O locations, the kernel module must call iomap_exit_shared_acc passing the iomap_acc_t value returned by iomap_enter_shared_acc.

RETURN VALUES

None

CONSTRAINTS

Can be called in user or interrupt context.
Can be called while holding spinlocks.

EXAMPLES

iomap_acc_t iomap_acc;

iomap_acc = iomap_enter_shared_acc();

/*
 * Read an I/O register that is shared with and
 * accessible by a user application.
 */

ioreg = *iop;

iomap_exit_shared_acc(iomap_acc);

SEE ALSO

iomap_enter_shared_acc(9F), kernel_iomap(9F), kernel_iounmap(9F), user_iomap(9F), user_iounmap(9F).
NAME
iomap_pagesize - Return pagesize in bytes for memory mapped I/O.

SYNOPSIS
#include <sys/iomap.h>
size_t iomap_pagesize(void);

PARAMETERS
None

DESCRIPTION
The iomap_pagesize function returns the pagesize in bytes for memory mapped I/O.

RETURN VALUES
Page size.

CONSTRAINTS
May be called in user or interrupt context.
May be called while holding spinlocks.

EXAMPLE
size_t pagesize;

/*
 * Verify the I/O physical address is aligned on a
 * page boundary in memory mapped I/O address space.
 */
pagesize = iomap_pagesize();
VASSERT(((uintptr_t)iophysaddr % pagesize) == 0);

SEE ALSO
kernel_iomap(9F), kernel_iounmap(9F), user_iomap(9F), user_iounmap(9F).
iovec(9S)

NAME

iovec – 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 1-2 iovec Structure Fields

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

iov_base The address of the user's buffer for uiomove or kernel buffer for WSIO CDIO
mapping services.

iov_len The number of bytes to be transferred.

SEE ALSO

uio(9F), wsio_fastmap(9F), wsio_map(9F), wsio_remap(9F)
NAME
kdaemon_thread - Kernel daemon thread routines.

SYNOPSIS
#include <sys/kdaemon_thread.h>

int kdaemon_thread_create(
    void (*func) (void *),
    void *data,
    struct proc *p,
    tid_t *newtid,
    uint64_t flags
);

void kdaemon_thread_exit(void);

int kdaemon_thread_wait(
    tid_t wait_for,
    tid_t *waited_for
);

DESCRIPTION
Kernel daemon threads, typically created at system initialization or DLKM initialization, have
no user context and run entirely within the kernel. In addition, they cannot respond to signals.
The kdaemon_thread_* API functions allow callers to create, manage, and destroy kernel
daemon threads.

The kdaemon_thread_* functions are intended to operate only upon kernel daemon threads.
Error returns or unspecified behavior might result if the kdaemon_thread_* functions are used

to act upon threads other than kernel daemon threads.

Summary of Available Calls
kdaemon_thread_create Creates a new kernel daemon thread within the specified
        process. The new kernel daemon thread begins execution at its
        start function.

kdaemon_thread_exit Causes the calling kernel daemon thread to exit.

kdaemon_thread_wait Allows a thread to wait for another kernel daemon thread.

See the individual kdaemon_thread_*(9F) manpages for a detailed description of each function.

EXAMPLES
/*
 * This code snippet has a main thread create a worker thread, which
 * does its work and then exits. The main thread then waits upon the
 * worker thread. The process p is a kernel daemon process.
 */
#include <sys/kdaemon_thread.h>

/*
 * Set up a worker thread to do work.
 */
ret = kdaemon_thread_create(work_thread,(void *)work,p,&newtid,
        (uint64_t)0);
if (ret != 0) {
    return ret;
} /*
 * Main thread does other processing.
 */

/*
 * Wait for the work_thread to complete.
 */
ret = kdaemon_thread_wait(newtid,NULL);

if (ret != 0) {
    return ret;
}

static void work_thread(void *work)
{
    /*
     * perform work
     */
    kdaemon_thread_exit();
}

SEE ALSO
kdaemon_thread_create(9F), kdaemon_thread_exit(9F), kdaemon_thread_wait(9F).
NAME

kdaemon_thread_create - Create a kernel daemon thread.

SYNOPSIS

#include <sys/kdaemon_thread.h>

int kdaemon_thread_create(
    void (*start_func) (void *),
    void *arg,
    struct proc *p,
    tid_t *newtid,
    uint64_t flags
);

PARAMETERS

start_func  Function in which the newly created kernel daemon thread will run. A return from this function will result in the termination of the kernel daemon thread created using this routine.

arg  The data passed as an argument to start_func upon startup of the newly created kernel daemon thread.

p  Process in which the newly created kernel daemon thread will be contained. This process can be a daemon process or a user process. However, if p is a user process, the calling process must also be p.

newtid  Pointer to address in which the thread ID of the newly created kernel daemon thread will be placed.

flags  The following values may be OR’ed together to specify the creation options for the created thread:

KDAEMON_THREAD_CREATE_DETACHED  Create a detached thread. This flag is automatically set if the process p is not a kernel daemon process.

KDAEMON_THREAD_CREATE_NO_WAIT  Create the thread only if it can be done without sleeping.

KDAEMON_THREAD_CREATE_NO_INTR_WAIT  Create the thread only if it can be done without sleeping interruptibly.

DESCRIPTION

Create a kernel daemon thread in a process. New kernel daemon threads can be created in both kernel daemon processes and user processes. Kernel daemon threads, typically created at system initialization or dlkm initialization, have no user context and run entirely within the kernel. Kernel daemon threads cannot respond to signals.

NOTE:  Kernel daemon threads created with this routine in user processes are created detached. If a kernel daemon thread is created in a different kernel daemon process without the KDAEMON_THREAD_CREATE_DETACHED flag set, that kernel daemon process must make arrangements to cleanup that kernel daemon thread’s resources if it exits.

The creation of kernel daemon threads in a user process different from the caller’s process is not supported.
RETURN VALUES

0 Success

EBUSY KDAEMON_THREAD_CREATE_NO_WAIT specified and the calling thread would have had to sleep waiting for a resource needed to complete this operation.

EBUSY KDAEMON_THREAD_CREATE_NO_INTR_WAIT specified and the calling thread would have had to sleep interruptibly waiting for a resource needed to complete this operation.

EINTR KDAEMON_THREAD_CREATE_NO_WAIT was not specified and a sleep waiting for a resource needed to complete this routine was interrupted.

EAGAIN The maximum number of threads in the containing process would be exceeded by a successful call to this routine.

ENOMEM No memory left to allocate space for a new kernel daemon thread.

EINVAL Invalid input parameters: Invalid flags, \( p \) is not a proc, \( newtid \) is NULL, or \( start_func \) is NULL.

EPERM An attempt was made to create a kernel daemon thread in user process \( p \) that is different from the calling process.

CONSTRAINTS

Do not call while holding a spinlock.

Do not call in an interrupt context.

EXAMPLES

/*
 * Create a thread that performs work and then exits implicitly.
 */
#include <sys/kdaemon_thread.h>
...
ret = kdaemon_thread_create(async_func,(void *)data,p,&newtid,
                         KDAEMON_THREAD_CREATE_DETACHED);
if (ret != 0) {
    return ret;
}
...

static void async_func(void *data)
{
    /*
     * Perform work.
     */
}

SEE ALSO

kdaemon_thread(9F), kdaemon_thread_exit(9F), kdaemon_thread_wait(9F).
**NAME**

kdaemon_thread_exit - Exit the currently running kernel daemon thread.

**SYNOPSIS**

```c
#include <sys/kdaemon_thread.h>
void kdaemon_thread_exit(void);
```

**PARAMETERS**

None

**DESCRIPTION**

This function causes the calling kernel daemon thread to exit. Only kernel daemon threads are to call this interface; the behavior of this function for non-daemon threads is undefined.

**RETURN VALUES**

None

**CONSTRAINTS**

Do not call while holding locks of any kind.
Do not call in interrupt context.

**EXAMPLES**

```c
/*
 * Within the start function used to start a kernel daemon thread,
 * the function exits when its work is complete.
 */
#include <sys/kdaemon_thread.h>
...
static void async_thread(void *data)
{
    while (1)
    {
        /*
         * Do new work.
         */
        if (work_done) {
            kdaemon_thread_exit();
        }
    }
}
```

**SEE ALSO**

kdaemon_thread(9F), kdaemon_thread_create(9F), kdaemon_thread_wait(9F).
kdaemon_thread_wait(9F)

NAME

kdaemon_thread_wait - Wait for a kernel daemon thread in the calling process to exit.

SYNOPSIS

#include <sys/kdaemon_thread.h>

int kdaemon_thread_wait(
    tid_t wait_for,
    tid_t *waited_for
);

PARAMETERS

wait_for Thread ID of the target kernel daemon thread to wait for. A thread ID of 0, specifiable only within kernel daemon processes, means to wait for any kernel daemon thread within the calling kernel daemon process to terminate.

waited_for Pointer to an address to store the thread ID of the thread that was waited for. This is useful if the calling kernel daemon process chose to wait for any kernel daemon thread (wait_for == 0). This can be NULL if this information is not desired.

DESCRIPTION

Wait for a kernel daemon thread within the calling process to terminate. You can only wait for non-detached kernel daemon threads (they were NOT created with the KDAEMON_THREAD_CREATE_DETACHED flag).

NOTE: Kernel daemon threads created with kdaemon_thread_create in user processes are automatically be created detached.

If tid is not 0, this call blocks until the kernel daemon thread corresponding to the tid terminates. If the tid parameter is 0 and the calling process is a kernel daemon process, it waits for the next thread in the kernel daemon process to terminate. If multiple threads wait for the same tid, at most one will return successfully, and the others return with an error of ESRCH.

RETURN VALUES

0 Success
ESRCH The kernel daemon thread specified by tid cannot be found in the calling process or is detached.
EPERM The thread corresponding to the thread ID is not a kernel daemon thread or wait_for is 0 and the calling process is not a kernel daemon process.
EINTR Wait was interrupted.
EDEADLK Performing this operation would result in system deadlock with this thread waiting on itself. This can happen, for instance, if a single-threaded kernel daemon process specifies a tid of 0.

NOTES

You can only wait for kernel daemon threads within your calling process. For example, if a thread creates a kernel daemon thread in another process, it will not be able to wait for that same kernel daemon thread.

CONSTRAINTS

Do not call while holding a spinlock.
EXAMPLES

/*
 * This code snippet creates a thread and then waits for it.
 */
#include <sys/kdaemon_thread.h>
...
ret = kdaemon_thread_create(func,(void *)data,p,&newtid,
   (uint64_t)0);
if (ret != 0) {
    return ret;
}

/*
 * Wait for the thread to complete.
 */
ret = kdaemon_thread_wait(newtid,NULL);
if (ret != 0) {
    return ret;
}
...

SEE ALSO

kdaemon_thread(9F), kdaemon_thread_create(9F), kdaemon_thread_exit(9F).
NAME

kernel_iomap - Map I/O memory to a kernel address.

SYNOPSIS

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

void *kernel_iomap(
    void *vaddr,
    void *paddr,
    pgcnt_t count,
    prot_t prot
);
```

PARAMETERS

- **vaddr**: Virtual address requested. must be NULL.
- **paddr**: Physical address of the I/O memory location from the view of the processor.
- **count**: Count of pages to map.
- **prot**: Protection mode for the mapping: PROT_KRW or PROT_URW.

DESCRIPTION

The `kernel_iomap` function memory maps into kernel address space a range of contiguous pages in I/O address space starting at the physical address `paddr` for `count` pages. The value returned is the kernel virtual address that maps onto the base of the I/O memory location specified by `paddr` and `count`.

The physical address is the 64-bit physical address from the view of the processor, and the address must be pagesize aligned. You can obtain the pagesize by calling `iomap_pagesize`.

The 64-bit physical address (`paddr`) is not necessarily the same address value that is obtained from the Base Address Register (BAR) of a PCI device. The caller might need to convert the BAR value to the appropriate 64-bit physical address value used by the processor.

**NOTE:** HP-UX 11i v3 kernels are 64-bit kernels and work with 64-bit physical address values. Although a processor might implement fewer than 64 bits of physical address space, the programming model views the physical address as a 64-bit value.

The `prot` value specifies the protection mode: PROT_KRW or PROT_URW. If set to PROT_KRW, the kernel is granted access to the I/O memory location, but no user application may be allowed access.

If set to PROT_URW, a user may be allowed to share access to the I/O memory location with the kernel. Obtain the user virtual address by calling `user_iomap`, passing the same `paddr` and `count` parameters.

**NOTE:** Access rights for the returned kernel virtual addresses in HP-UX 11i v3 are always set to PROT_KRW, regardless of the value passed to `prot`.

To successfully map an I/O memory range multiple times, pass the same `paddr`, `count`, and `prot` values in each call to `kernel_iomap`.

If a driver does not intend to share access to I/O memory with any user applications, it must call `kernel_iomap` with PROT_KRW, which restricts access to the kernel only. If the driver calls `kernel_iomap` with PROT_KRW, a subsequent call to `user_iomap` or `user_iomap_private` for the same I/O memory will fail.
You cannot change a mapping with PROT_KRW by calling kernel_iomap with PROT_URW. You must first remove the mapping in order to change the protection. If only user applications will access I/O memory and the driver will never access the I/O memory, the driver only needs to call user_iomap or user_iomap_private; it does not need to call kernel_iomap.

**RETURN VALUES**

<>NULL Kernel virtual address.
NULL Error. Mapping failed.

**CONSTRAINTS**

Do not call in interrupt context.
Do not call while holding a spinlock.

**EXAMPLES**

```c
void  *physaddr;
void  *useraddr;
volatile int   *iop;
size_t  pagesize;
int    ioreg;

/*
 * Get the physical address as seen by the processor.
 * Verify that the physical address is pagesize aligned.
 */
physaddr = xyz;
pagesize = iomap_pagesize();
VASSERT(((uintptr_t)physaddr % pagesize) == 0);

/*
 * Memory map the kernel virtual address and set the
 * protection on the pages to also allow user access.
 */
iop = kernel_iomap(NULL, physaddr, 1L, PROT_URW);
if (iop == NULL) {
    /*
     * Failed to map kernel virtual address.
     * Add error handling code here.
     */
}

/*
 * Now memory map the user virtual address using the
 * same physical address and page count parameters.
 */
useraddr = user_iomap(NULL, physaddr, 1L);
if (useraddr == NULL) {
    /*
     * Failed to map user virtual address.
     * Add error handling code here.
     */
}

/*
 * Access an IO register in the memory mapped page.
 */
ioreg = *iop;
```
SEE ALSO
iomap_enable_zwc(9F), iomap_pagesize(9F), kernel_iounmap(9F), user_iomap(9F), user_iomap_private(9F), user_iounmap(9F).
**NAME**

kernel_iomap_public - Publicly map I/O memory to a kernel address.

**SYNOPSIS**

```c
#include <sys/iomap.h>
void *kernel_iomap_public(
    void *vaddr,
    void *paddr,
    pgcnt_t count,
    prot_t prot
);
```

**PARAMETERS**

- **vaddr** Virtual address requested. must be NULL.
- **paddr** Physical address of the I/O memory location from the view of the processor.
- **count** Count of pages to map.
- **prot** Protection mode for the mapping: PROT_KRW or PROT_URW.

**DESCRIPTION**

The `kernel_iomap_public` function is a deprecated interface. The recommended interface is `kernel_iomap`.

Use of `kernel_iomap_public` is not advised because it creates an unprotected mapping for the device I/O memory. If mapped with `kernel_iomap_public` and PROT_URW, any 64-bit process that can form the mapped virtual address will be able to gain access to the I/O memory since the associated pages are mapped with public access protection (that is, with no protection key) and user privilege level access.

The `kernel_iomap_public` function memory maps into kernel address space a range of contiguous pages in I/O address space starting at the physical address `paddr` for `count` pages. The value returned is the kernel virtual address that maps onto the base of the I/O memory location specified by `paddr` and `count`.

The physical address is the 64-bit physical address from the view of the processor, and the address must be pagesize aligned. The pagesize can be obtained by calling `iomap_pagesize`.

The 64-bit physical address `paddr` is not necessarily the same address value that is obtained from the Base Address Register (BAR) of a PCI device. The caller may need to convert the BAR value to the appropriate 64-bit physical address value used by the processor.

The `prot` value specifies the protection mode: either PROT_KRW or PROT_URW. If set to PROT_KRW, protection on the memory mapped pages is set such that the kernel is granted access to the I/O memory location, but no user application may be allowed access.

If set to PROT_URW, the protection is set such that a user may be allowed to share access with the kernel. The user virtual address is obtained by calling `user_iomap` passing the same `paddr` and `count` parameters.

**RETURN VALUES**

- **<>NULL** Kernel virtual address.
- **NULL** Error. Mapping failed.

**CONSTRAINTS**

Do not call in an interrupt context.
Do not call while holding a spinlock.

**EXAMPLES**

```c
void  *physaddr;
void  *useraddr;
int   *iop;
sizet pagesize;
int ioreg;

/*
 * Get the physical address as seen by the processor.
 * Verify that the physical address is pagesize aligned.
 */
physaddr = xyz;
pagesize = iomap_pagesize();
VASSERT(((uintptr_t)physaddr % pagesize) == 0);

/*
 * Publicly map the kernel virtual address and set the
 * protection on the pages to also allow user access.
 * WARNING: Public mapping allows all users, including
 * unauthorized and errant user applications,
 * to access the I/O memory location.
 */
iop = kernel_iomap_public(NULL, physaddr, 1L, PROT_URW);

/*
 * Now memory map the user virtual address using the
 * same physical address and page count parameters.
 */
useraddr = user_iomap(NULL, physaddr, 1L);

/*
 * Access the I/O memory location.
 */
oireg = *iop;
```

**SEE ALSO**

iomap_pagesize(9F), kernel_iomap(9F), kernel_iounmap(9F), user_iomap(9F), user_iomap_private(9F),
user_iounmap(9F).
kernel_iounmap(9F)

NAME

kernel_iounmap - Unmap I/O memory for the assigned kernel address.

SYNOPSIS

#include <sys/iomap.h>

int kernel_iounmap(
    void *vaddr,
    pgcnt_t count
);

PARAMETERS

vaddr Virtual address returned by kernel_iomap.

count Count of pages to unmap.

DESCRIPTION

The kernel_iounmap function removes the kernel virtual address assigned to map a range of I/O memory pages.

The vaddr parameter is the virtual address returned by kernel_iomap, and the count parameters must match the count parameter passed to kernel_iomap.

RETURN VALUES

0 Success.

1 Failure.

CONSTRAINTS

Do not call in an interrupt context.

Do not call while holding a spinlock.

EXAMPLES

void    *physaddr;
void    *kernaddr;
pgcnt_t count;
int     retval;

...

/*
 * Memory map the kernel virtual address onto the I/O memory
 * location for count number of pages.
 */
kernaddr = kernel_iomap(NULL, physaddr, count, PROT_KRW);

...

/*
 * Remove the mapping for the kernel virtual address.
 */
retval = kernel_iounmap(kernaddr, count);
VASSERT(retval == 0);

SEE ALSO

iomap_pagesize(9F), kernel_iomap(9F), user_iomap(9F), user_iomap_private(9F). user_iounmap(9F).
NAME

kfree - 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

None

SEE ALSO

kmalloc(9F)
NAME

kmalloc - 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 is always contiguous in the virtual address space, but might be discontiguous in the physical address space. If size is greater than PAGESIZE, the physical pages allocated might not be physically contiguous.

Drivers must 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 (IHV) must specify the M_IHV pool for general I/O purposes of their drivers.

The flags parameter can optionally have the M_NOWAIT flag set. If M_NOWAIT is set and no memory is available from the requested pool type, the function returns 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 either:

• Called in an interrupt context
• 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 must 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 following conditions must be true:

• The caller must not be in an interrupt context
• No spinlocks are held

SEE ALSO

kfree(9F)
NOTES

It is generally a bad programming practice to allocate memory within critical sections of your code. The kernel memory allocator, including the \texttt{kmalloc} interface, might be required to do a lot of work to satisfy an allocation request, and will sleep to wait for memory to become available, if necessary. This can have a large negative impact on the performance of your code.

There are two critical sections in which sleeping is prohibited by the kernel:

- Interrupt routines (code running in interrupt context)
- Code executed while holding a spinlock.

The \texttt{M\_NOWAIT} flag enables you to allocate memory in these contexts. If \texttt{M\_NOWAIT} is specified, the kernel memory allocator only tries the fast paths of memory allocation, returning \texttt{NULL} if unsuccessful. In this case, your code must be prepared to deal with a memory allocation failure. Typically, this requires retrying the allocation outside the critical section with the \texttt{M\_WAITOK} flag specified.

HP strongly recommends that whenever possible you structure your code so all memory allocations are done outside of critical sections.
NAME

kmem_arena_alloc - Allocates a fixed size memory object from the arena.

SYNOPSIS

#include <sys/vm_arena_iface.h>

void *kmem_arena_alloc(
    kmem_handle_t handle,
    arena_flags_t flags
);

PARAMETERS

handle Opaque arena handle returned by kmem_arena_create.
flags Allocation flags.

DESCRIPTION

The kmem_arena_alloc function allocates a memory object from the specified arena. This function is called for arenas created with fixed sized allocations.

The handle parameter is the value returned by kmem_arena_create. It represents an arena created with fixed sized memory objects.

The flags parameter may contain one of the following values:

M_WAITOK Allow the thread to block and wait for memory allocation.
M_NOWAIT Do not allow the thread to block and wait. The function returns NULL if memory allocation will block the thread (for example, put the thread to sleep).

RETURN VALUES

<> NULL Success: Pointer to the memory object allocated.
NULL Failure: Unable to allocate a memory object.

CONSTRAINTS

Do not call while holding a spinlock.
Do not call in an interrupt context if flags is set to M_WAITOK.

EXAMPLES

The following example allocates a fixed sized memory object from the arena.

```c
kmem_arena_handle_t my_arena;
void *objp;

/* Create a fixed sized arena with default attributes. */
my_arena = kmem_arena_create(sizeof(my_type), "MY_ARENA_NAME", NULL, M_WAITOK);

/* Allocate a fixed sized memory object from the arena. */
objp = kmem_arena_alloc(my_arena, M_NOWAIT);

/* Allocation can fail with M_NOWAIT, so check the return value. */
if (objp == NULL) {
    /* Put error handling code here. */
```
SEE ALSO

kmem_arena_create(9F), kmem_arena_destroy(9F), kmem_arena_free(9F), kmem_arena_init(9F),
kmem_arena_varalloc(9F)

NOTES

It is generally a bad programming practice to allocate memory within critical sections of your
code. The kernel memory allocator, including the kmem_arena_alloc interface, might be
required to do a lot of work to satisfy an allocation request, and will sleep to wait for memory
to become available, if necessary. This can have a large negative impact on the performance of
your code.

There are two critical sections in which sleeping is prohibited by the kernel:
• Interrupt routines (code running in interrupt context)
• Code executed while holding a spinlock.

The M_NOWAIT flag enables you to allocate memory in these contexts. If M_NOWAIT is specified,
the kernel memory allocator only tries the fast paths of memory allocation, returning NULL if
unsuccessful. In this case, your code must be prepared to deal with a memory allocation failure.
Typically, this requires retrying the allocation outside the critical section with the M_WAITOK
flag specified.

HP strongly recommends that whenever possible you structure your code so all memory
allocations are done outside of critical sections.
NAME

kmem_arena_attr_init - Initializes the arena attributes in the kmem_arena_attr_t structure.

SYNOPSIS

#include <sys/vm_arena_iface.h>

void kmem_arena_attr_init(
    kmem_arena_attr_t *kattr,
    size_t size
);

PARAMETERS

kattr Pointer to the kmem_arena_attr_t data structure.
size Size in bytes of the kmem_arena_attr_t data structure.

DESCRIPTION

The kmem_arena_attr_init function initializes the kmem_arena_attr_t structure that is passed to kmem_arena_create. The function offloads from the caller the burden of initializing all the elements in the structure. Normally, only a few elements, if any, need to be modified by the caller.

The size parameter specifies the size of the kmem_arena_attr_t structure. This allows binary compatibility with future extensions to the kmem_arena_attr_t structure. If new attributes are added, the caller is not required to recompile. The size parameter must be passed as sizeof(kmem_arena_attr_t).

The elements of the kmem_arena_attr_t structure are defined as follows:

- **kat_struct_size** Specifies the size of the structure passed by the caller to kmem_arena_attr_init. It enables future extensions to the kmem_arena_attr_t structure without breaking binary compatibility with the caller. Do not modify this element.

- **kat_ctor()** Pointer to the constructor function, ctor, which does the work of initializing memory objects added to refill the arena. The constructor is intended to implement object caching.

- **kat_dtor()** Pointer to the destructor function, dtor, which does the work of dismantling memory objects reclaimed from the arena. The destructor is intended to implement object caching.

- **kat_maxcnt** Specifies the maximum number of memory objects allocated for the arena. This is used only for fixed size memory objects.

- **kat_minfcnt** Specifies the minimum number of memory objects per SPU to be kept on the free list of the arena. This is used as an advisory for managing small objects.

- **kat_maxpgcnt** Specifies the maximum number of 4K pages allocated for the arena. This is used only for variable sized memory objects.

- **kat_refillcnt** Specifies the number of objects to be added to the free list per refill.

- **kat_flags** Specifies special attribute flags, which are defined as follows:
  - KAT_NO_LGPG The memory objects allocated from the arena will not use large (super) pages. Allocations will only use 4K pages, which is
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KAT_ALLOC32</td>
<td>The memory objects allocated from the arena will have a 32-bit virtual address. This attribute is applicable only for PA-RISC implementations and is not recommended for general use.</td>
</tr>
<tr>
<td>KAT_CACHE_XLARGE_OBJECTS</td>
<td>The arena caches xlarge memory objects. By default, the arena will not cache xlarge objects to avoid memory fragmentation, but some modules that use xlarge objects may want to cache them to improve performance.</td>
</tr>
<tr>
<td>KAT_ALIGN_ON_SIZE</td>
<td>The memory objects allocated from the arena will be aligned on a power-of-two address. For example, a size of 200 bytes will be aligned on a 256 byte boundary. Similarly, a size of 6000 bytes will be aligned on 8K boundary.</td>
</tr>
<tr>
<td>KAT_ALIGN_ON_SIZE_COMPAT</td>
<td>This is to provide similar alignment as the deprecated MALLOC interface. Sizes that are less than 4K are aligned on size but sizes greater than or equal to 4K are aligned on 4K. For example, a size of 200 bytes will be aligned on a 256 byte boundary but a size of 6000 bytes is aligned on 4K.</td>
</tr>
<tr>
<td>KAT_CONTIGUOUS_PHYSMEM</td>
<td>The memory objects allocated from the arena will be contiguous in physical memory.</td>
</tr>
<tr>
<td>KAT_MULTICACHE_SIZE</td>
<td>The memory objects allocated from the arena will be a multiple of cacheline size. This is required by certain drivers for DMA transactions.</td>
</tr>
<tr>
<td>KAT_DMA32</td>
<td>The memory objects allocated from the arena will have a 32-bit physical address. This attribute is applicable only for IA64 implementations. It is required by drivers that control devices limited to 32-bit DMA addressing.</td>
</tr>
</tbody>
</table>

**kat_align** Specifies the alignment of the object in the arena. It is used only for fixed size memory objects. The alignment for small objects must be a multiple of eight bytes and be a power of two. The alignment of objects greater than or equal to 4K are by default 4K-aligned and cannot be changed.
RETURN VALUES
None

CONSTRAINTS
May be called in user or interrupt context.
May be called while holding a spinlock.

EXAMPLES
The following example creates a variable sized arena.

```c
kmem_arena_attr_t attr;
kmem_arena_handle_t my_arena;

/*
 * Initialize the arena attributes to default values.
 */
kmem_arena_attr_init(&attr, sizeof(kmem_arena_attr_t));
/*
 * Alter the default arena attributes to align memory objects
 * on a power-of-two address. The power-of-two value is greater
 * than or equal to the size of the memory object allocated.
 */
attr.kat_flags |= KAT_ALIGN_ON_SIZE;
/*
 * Create my variable sized arena.
 */

my_arena = kmem_arena_create(0, "MY_ARENA_NAME", &attr, M_WAITOK);
```

SEE ALSO
kmem_arena_alloc(9F), kmem_arena_create(9F), kmem_arena_destroy(9F), kmem_arena_free(9F), kmem_arena_varalloc(9F).
**NAME**

`kmem_arena_create` - Creates and initializes an arena.

**SYNOPSIS**

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

void *kmem_arena_create(
    size_t size,
    char *arena_name,
    kmem_arena_attr_t *kattr,
    arena_flags_t flags
);
```

**PARAMETERS**

- `size` Size in bytes of the memory objects allocated from the arena.
- `arena_name` Name of the arena, which must be unique across the system.
- `kattr` Pointer to the `kmem_arena_attr_t` structure.
- `flags` Creation flags.

**DESCRIPTION**

The `kmem_arena_create` function creates and initializes an arena, which is a user defined pool of memory objects. Separate arenas must be created for unrelated memory objects that are frequently allocated. A single arena may be created for all infrequently allocated memory objects with similar attributes.

The `size` parameter specifies the size of memory objects allocated from the arena. If `size` is the value zero, the arena will allocate variable sized memory objects; otherwise, the arena will allocate memory objects of a fixed size. Variable sized memory objects are allocated by calling `kmem_arena_varalloc` and fixed sized memory objects are allocated by calling `kmem_arena_alloc`.

The `arena_name` parameter points to a zero-terminated character string that identifies the arena by name. Choose a descriptive name that will be unique across the system.

Attributes that describe the arena are passed through the `kattr` parameter. If this parameter is `NULL`, default attributes are applied to the arena. See `kmem_arena_attr_init(9F)` for information on how to initialize arena attributes.

The `flags` parameter may contain one of the following flag values:
- `M_WAITOK` Allow the thread to block and wait while creating the arena.
- `M_NOWAIT` Do not allow the thread to block and wait. The function returns `NULL` if creation of the arena will block the thread (for example, put the thread to sleep).

If successful, the function returns an opaque arena handle, which is used for all allocations and de-allocations of memory objects from and to the arena.

**RETURN VALUES**

- `<> NULL` Success: Opaque arena handle.
- `NULL` Failure: No arena allocated.

**CONSTRAINTS**

Do not call while holding a spinlock.

Do not call in interrupt context if `flags` is set to `M_WAITOK`. 
EXAMPLES

The following example creates an arena with fixed sized memory objects and default attributes.

```c
kmem_arena_attr_t attr;
kmem_handle_t var_arena;
kmem_handle_t fixed_arena;

/*
 * Specify memory objects are to be aligned on a cacheline.
 */
kmem_arena_attr_init(&attr, sizeof(kmem_arena_attr_t));
attr.kat_flags |= KAT_MULTICACHE_SIZE;
/*
 * Create an arena with variable sized memory objects, where each
 * memory object is physically aligned on a cacheline.
 */
var_arena = kmem_arena_create(0,
    "MY_VARIABLE_ARENA_NAME", &attr, M_WAITOK);
/*
 * Create an arena with fixed sized memory objects and default attributes.
 */
fixed_arena = kmem_arena_create(sizeof(my_type),
    "MY_FIXED_ARENA_NAME", NULL, M_WAITOK);
```

SEE ALSO

kmem_arena_alloc(9F), kmem_arena_destroy(9F), kmem_arena_free(9F), kmem_arena_init(9F),
kmem_arena_varalloc(9F).
kmem_arena_destroy(9F)

NAME

kmem_arena_destroy - Destroys the arena.

SYNOPSIS

#include <sys/vm_arena_iface.h>
kmem_handle_t kmem_arena_destroy(
    kmem_handle_t handle
);

PARAMETERS

handle Opaque arena handle returned by kmem_arena_create.

DESCRIPTION

The kmem_arena_destroy function destroys the arena created by kmem_arena_create. All memory objects allocated from the arena must be freed before calling this function, and subsequent requests to allocate memory objects in the arena will fail.

RETURN VALUES

NULL This function always returns NULL. The caller can use this routine to reset the handle to the arena just freed.

CONSTRAINTS

Can be called in user or interrupt context.
Do not call while holding a spinlock.

EXAMPLES

The following example destroys the arena once it is no longer needed.

kmem_arena_handle_t my_arena;
    /* Create a variable sized arena with default attributes.
     */
    my_arena = kmem_arena_create(0, "MY_ARENA_NAME", NULL, M_WAITOK);
    ... 
    /*
     * Destroy the arena now, as it is no longer needed.
     */
    my_arena = kmem_arena_destroy(my_arena);

SEE ALSO

kmem_arena_alloc(9F), kmem_arena_create(9F), kmem_arena_free(9F), kmem_arena_init(9F), kmem_arena_varalloc(9F)
NAME

kmem_arena_free - Frees a memory object to its associated arena.

SYNOPSIS

#include <sys/vm_arena_iface.h>

void *kmem_arena_free(
    void *objp,
    arena_flags_t flags
);

PARAMETERS

objp Pointer to the memory object to be freed.
flags Free flags.

DESCRIPTION

The kmem_arena_free function frees a memory object and returns it to its associated arena. Once the memory object is freed, it cannot be accessed. The memory object may have been allocated by either kmem_arena_alloc or kmem_arena_varalloc.

The flags parameter may contain one of the following flag values:
M_WAITOK Allow the thread to block and wait while freeing the memory object.
M_NOWAIT Do not allow the thread to block and wait.

RETURN VALUES

NULL This function always returns NULL. It can be used by the caller to reset the pointer to the memory object just freed.

CONSTRAINTS

Do not call while holding a spinlock.
Do not call in an interrupt context if flags set to M_WAITOK.

EXAMPLES

The following example frees the memory object that is no longer used.

kmem_arena_handle_t my_arena;
void *objp;

    /*
     * Create a variable sized arena with default attributes.
     */
    my_arena = kmem_arena_create(0, "MY_ARENA_NAME", NULL, M_WAITOK);
    ...
    /*
     * Allocate a variable sized memory object from the arena.
     */
    objp = kmem_arena_varalloc(my_arena, sizeof(my_type), M_NOWAIT);
    ...
    /*
     * Free the memory object now that we are done using it.
     */
    objp = kmem_arena_free(objp, M_NOWAIT);
SEE ALSO

kmem_arena_alloc(9F), kmem_arena_create(9F), kmem_arena_destroy(9F), kmem_arena_init(9F), kmem_arena_varalloc(9F)
kmem_arena_varalloc(9F)

NAME

kmem_arena_varalloc - Allocate a variable sized memory object from the arena.

SYNOPSIS

#include <sys/vm_arena_iface.h>

void *kmem_arena_varalloc(
    kmem_handle_t handle,
    size_t size,
    arena_flags_t flags
);

PARAMETERS

handle   Opaque arena handle returned by kmem_arena_create.
size     Size in bytes of requested memory object.
flags    Allocation flags.

DESCRIPTION

The kmem_arena_varalloc function allocates a memory object from the specified arena. This function is called for arenas created with variable sized allocations.

The handle parameter is the value returned by kmem_arena_create. It represents an arena created with variable sized memory objects.

The size parameter is the requested size in bytes of the memory object to be allocated.

The flags parameter may contain one of the following flag values:

M_WAITOK    Allow the thread to block and wait for memory allocation.
M_NOWAIT    Do not allow the thread to block and wait. The function returns NULL if memory allocation will block the thread (for example, put the thread to sleep).

RETURN VALUES

<> NULL   Success: Pointer to the memory object allocated.
NULL      Failure: Unable to allocate a memory object.

CONSTRAINTS

Do not call while holding a spinlock.

Do not call in interrupt context if flags is set to M_WAITOK.

EXAMPLES

The following example allocates a variable sized memory object from the arena.

```
kmem_arena_handle_t my_arena;
void *objp;

/*
 * Create a variable sized arena with default attributes.
 */
my_arena = kmem_arena_create(0, "MY_ARENA_NAME", NULL, M_WAITOK);

/*
 * Allocate a variable sized memory object from the arena.
 */
objp = kmem_arena_varalloc(my_arena, sizeof(my_type), M_NOWAIT);
/*
```
* Allocation can fail with M_NOWAIT, so check the return value.
 */
if (objp == NULL) {
  /*
   * Put error handling code here.
   */
}

SEE ALSO
kmem_arena_alloc(9F), kmem_arena_create(9F), kmem_arena_destroy(9F), kmem_arena_free(9F), kmem_arena_init(9F)

NOTES
It is generally a bad programming practice to allocate memory within critical sections of your code. The kernel memory allocator, including the kmem_arena_varalloc interface, might be required to do a lot of work to satisfy an allocation request, and will sleep to wait for memory to become available, if necessary. This can have a large negative impact on the performance of your code.

There are two critical sections in which sleeping is prohibited by the kernel:
• Interrupt routines (code running in interrupt context)
• Code executed while holding a spinlock.

The M_NOWAIT flag enables you to allocate memory in these contexts. If M_NOWAIT is specified, the kernel memory allocator only tries the fast paths of memory allocation, returning NULL if unsuccessful. In this case, your code must be prepared to deal with a memory allocation failure. Typically, this requires retrying the allocation outside the critical section with the M_WAITOK flag specified.

HP strongly recommends that whenever possible you structure your code so all memory allocations are done outside of critical sections.
NAME
kr_close_node - Close a registry key and free all resources.

SYNOPSIS
#include <sys/krs.h>
int kr_close_node(
     kr_key_t key
);  

PARAMETERS
key The KRS key representing the node that is to be closed.

DESCRIPTION
The kr_close_node routine closes a node, previously opened by a call to kr_open_node or kr_link_node. Upon successful completion, all resources associated with the open instance are freed; the key is no longer valid.

RETURN VALUES
KR_ERR_BADKEY The specified key does not exist.
KR_SUCCESS Request completed successfully.

CONSTRAINTS
Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES
int kr_open_node(kr_key_t, char *, kr_flags_t, kr_key_t*);
int kr_close_node(kr_key_t);myfunc()
{
    int rv;
k_key_t mykey;
    rv = kr_open_node(KR_NOKEY, "/mynode", KR_CREATE, &mykey);
    if (rv != KR_SUCCESS) {
        return ERROR;
    }
    ...
k_close_node(mykey);
}

SEE ALSO
kr_delete_node(9F), kr_link_node(9F), kr_open_node(9F).
kr_delete_node(9F)

NAME

kr_delete_node - Delete the given registry tree node or branch.

SYNOPSIS

int kr_delete_node(  
    kr_key_t key,  
    kr_flags_t flags
);

PARAMETERS

key The KRS key representing the node or branch to be deleted.
flags The flags parameter values are as follows:
    KR_ALL If the target node is not a leaf node, delete the entire branch.
    KR_LINKS If any of the deleted nodes have links, delete all of the links associated with them.

DESCRIPTION

The kr_delete_node routine causes one or more nodes to be deleted from the KRS tree. If the KR_ALL flag is specified, the entire subtree starting at the node is deleted. If the KR_ALL flag is not specified, and the node has children, and there are no links to the node, the request fails. If the KR_LINKS flag is specified, all links to any of the deleted nodes are also deleted.

Deleting a node does not close the key. This is because a key to an open node can be shared between multiple threads of execution. Because there is no mechanism for identifying these threads to notify them of the deletion, the node must remain open until an explicit kr_close_node is performed. If a thread tries to use the key after it is deleted, but before it has been closed, the function returns KR_ERR_DELETED to inform it of the current state. After the node has been closed, the function returns KR_ERR_BADKEY; the key is no longer recognized by the system.

RETURN VALUES

KR_ERR_BADKEY The specified key does not exist.
KR_ERR_DELETED The node represented by the given KRS key has been deleted since the node was opened.
KR_ERR_NOTLEAF Target node is not a leaf node (if KR_ALL is not specified).
KR_ERR_PARAM Attempt to delete the root node.
KR_ERR_RDONLY An attempt was made from user-space to delete a read-only node or the child of a read-only node.
KR_SUCCESS Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

int kr_delete_node(kr_key_t, kr_flags_t);myfunc()
{
    kr_key_t mykey;/* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey);
    kr_delete_node(mykey, KR_NOFLAGS);
    kr_close_node(mykey);}
SEE ALSO

kr_close_node(9F), kr_link_node(9F), kr_open_node(9F).
kr_delete_value(9F)

NAME

kr_delete_value - Delete the specified named value from an open node.

SYNOPSIS

#include <sys/krs.h>

int kr_delete_value(
    kr_key_t key,
    char *vname,
    kr_flags_t flags
);

PARAMETERS

key The KRS key representing the node from which the value is to be deleted.
vname The name of the value to delete.
flags The valid value is KR_NOFLAGS.

DESCRIPTION

The kr_delete_value routine deletes the specified value from the node identified by key.

RETURN VALUES

KR_ERR_BADKEY The specified key does not exist.
KR_ERR_DELETED The node represented by the given KRS key has been deleted since the node was opened.
KR_ERR_NAME The length of the specified value name exceeds KR_NAME_LEN.
KR_ERR_NOTFOUND The specified value could not be found.
KR_ERR_RDONLY An attempt was made from user-space to delete a read-only value or delete a value in a read-only node.
KR_SUCCESS Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

int kr_delete_value(kr_key_t, char *, kr_flags_t);

myfunc()
{
    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey);
    kr_delete_value(mykey, "v0", KR_NOFLAGS);
    kr_close_node(mykey);
}

SEE ALSO

kr_get_value(9F), kr_open_node(9F), kr_set_value(9F).
**NAME**

kr_flush - Cause KRS data to be saved to persistent storage.

**SYNOPSIS**

#include <sys/krs.h>

int kr_flush(void);

**PARAMETERS**

None

**DESCRIPTION**

Normally, changes made to persistent data in KRS are saved to disk at predetermined time intervals. If you call this routine, the system saves the data to disk as soon as possible.

The kr_flush routine sends a signal to the system daemon responsible for saving KRS data and returns as soon as the signal is sent; it does not wait for the data to be written to disk (non-blocking). Use the kr_get_mod_time routine to determine when the data is actually saved to disk.

To maximize performance efficiency, do not call kr_flush until you are done changing persistent data.

**RETURN VALUES**

KR_FAIL The KRS daemon was not running and the signal could not be sent. This must be a temporary situation as the daemon is respawned by init. If the signal is not being sent the daemon saves the data as soon as it is respawned.

KR_SUCCESS The signal to the daemon was sent successfully.

**EXAMPLES**

int kr_flush();

myfunc()
{
    kr_key_t key;
    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_CREATE |
                 KR_PERSISTENT, &key);
    kr_close_node(key);
    kr_flush();
}

**SEE ALSO**

kr_get_mod_time(9F), kr_open_node(9F).
NAME

kr_get_mod_time - Return the last time persistent data was modified in KRS.

SYNOPSIS

#include <sys/krs.h>

int kr_get_mod_time(
    kr_flag_field_t class,
    kr_flag_field_t phase,
    uint64_t *mtime
);

PARAMETERS

class  The specific class of persistent data that is relevant to the request. The valid values are as follows:

KR_CLASS_KERNEL      Return the last time kernel-specific persistent data was changed (governed by phase).
KR_CLASS_SYSTEM      Return the last time system-specific persistent data was changed (governed by phase).
KR_CLASS_SYNC        Return the last time KRS data was flushed to persistent storage. When specified, phase is ignored.

phase    The phase within the appropriate class. This must be KR_PHASE1 or KR_PHASE_ANY.

mtime    A pointer to caller-supplied storage, through which the modification time is returned.

DESCRIPTION

The kr_get_mod_time routine returns a time stamp indicating the last time persistent data matching the requested class and phase was modified. Use the returned time stamps only for relative comparisons. If time stamp B is greater than time stamp A, time stamp B indicates a more recent change.

RETURN VALUES

KR_ERR_PARAM    Either an unrecognized class or unrecognized phase was specified.
KR_SUCCESS      The request completed successfully.

CONSTRAINTS

None

EXAMPLES

int kr_get_mod_time(kr_flag_field_t, kr_flag_field_t, uint64_t);
int kr_get_file_name(char *, kr_size_t, kr_flag_field_t, kr_flag_field_t);
void kr_print_tree(kr_key_t, char *, int, int);
myfunc()
{
    /* Error checking and local declarations omitted */
    kr_get_mod_time(KR_CLASS_ANY, KR_PHASE_ANY, &time);
    kr_get_file_name(name_buf, &buf_sz, KR_FILE_SYS,
                KR_PHASE1);
    kr_print_tree(KR_NOKEY, "/", 0, 0);
}
SEE ALSO

kr_flush(9F), kr_open_node(9F).
NAME

kr_get_node_info - Get information about a given node.

SYNOPSIS

#include <sys/krs.h>

int kr_get_node_info(
    kr_key_t key,
    kr_flag_field_t *info_buf,
    kr_flag_field_t *vtyp_buf,
    kr_flag_field_t vclass,
    kr_flag_field_t vphase,
    kr_flag_field_t *flag_buf,
    kr_linkid_t *lid_buf
);

PARAMETERS

key The KRS key representing the node whose information is to be obtained.
info_buf A pointer to client-supplied storage through which informational flags are returned. These flags convey the following:
    KR_HASCHILDREN The node in question is not a leaf node. It has children.
    KR_LINKED The data associated with the node in question is linked to by at least one other node.

vtyp_buf A pointer to client-supplied storage through which bit fields indicating the types of values contained within the node are returned.

vclass Value class. When collecting information about the node's values, only include information about the values that match vclass, as follows:
    KR_CLASS_NONE Do not return any value information for the node.
    KR_CLASS_KERNEL Only return information for values that are flagged kernel-specific.
    KR_CLASS_SYSTEM Only return information for values that are flagged system-specific.
    KR_CLASS_ANY Return information for both kernel- and system-specific values.

vphase Value phase. When collecting information about the node's values, only include information about values matching vphase, as follows:
    KR_PHASE1 Match Phase 1 persistent values.
    KR_PHASE_ANY Match any persistent value.

flag_buf A pointer to client-supplied storage through which the attribute flags associated with the node in question are returned.

lid_buf A pointer to client-supplied storage through which the link ID associated with the node's data is returned. This link ID uniquely identifies the data portion of the node and is the same for all nodes linked to the data.

DESCRIPTION

The kr_get_node_info routine returns information specific to the given node. The information returned is as described in the Parameters section.
RETURN VALUES

KR_ERR_BADKEY   The specified key does not exist.
KR_ERR_DELETED  The node represented by the given KRS key has been deleted since the node was opened.
KR_SUCCESS      Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

int kr_get_node_info(kr_key_t, kr_flag_field_t, kr_flag_field_t,
                     kr_flag_field_t, kr_flag_field_t,
                     kr_flag_field_t, kr_link_id_t);
int kr_get_vinfo(kr_key_t, char *, kr_type_t *, kr_size_t *,
                 kr_flag_field_t *);

myfunc()
{
    /* Error checking and local declarations omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &key);
    kr_get_node_info(key, &info_buf, &vtype_buf, vclass,
                     vphase, &flag_buf, &lid_buf);
    kr_get_vinfo(key, "v0", &type_buf, &size_buf, &vflag_buf);
    kr_close_node(key);
}

SEE ALSO

kr_get_node_names(9F), kr_open_node(9F), kr_set_node_flags(9F).
kr_get_node_names(9F)

NAME
kr_get_node_names - Retrieve a list of node names, as specified by flags.

SYNOPSIS
#include <sys/krs.h>

int kr_get_node_names(
    kr_key_t key,
    char *names[],
    int *n_name,
    char *buf,
    kr_size_t *buf_sz,
    kr_flags_t flags
);

PARAMETERS
key The KRS key representing the node relative to which node names are to be retrieved.
names A client-supplied array through which pointers to the node names are returned.
n_name When called, the number of elements in the names array. On return, the actual number
    of names that have been returned or the required size of the names array if it is not
    large enough to fulfill the request.
buf A client-supplied buffer through which the text of the node names are returned.
    Pointers returned in the names array point into this buffer.
buf_sz When called, the size of the buffer pointed to by buf. On return, the actual number of
    characters returned through buf, or the required size of the buffer if it is not large
    enough to fulfill the request.
flags The flags parameter values are as follows:
    KR_PATH Return the names of all the nodes in the path to the target node.
    Otherwise, return the names of the target node’s children.

DESCRIPTION
The kr_get_node_names routine returns node names as governed by flags. If the KR_PATH
flag is specified, the names of the nodes comprising the path to the target node are returned.
Otherwise, the names of the node’s children are returned.

The names are returned through the names and buf parameters. The buf parameter points to a
caller-supplied buffer that will contain the text of the names being returned. The names parameter
points to a caller-supplied array of character pointers that will contain pointers to each name in
buf. The n_name and buf_sz parameters point to caller-supplied storage containing the size of the
names array and buf buffer respectively.

If the storage supplied by the caller is not enough to fulfill the request, the routine sets n_name
and buf_sz to indicate the storage required; the request fails with KR_ERR_SIZE. Upon successful
completion, n_name is set to the actual number of names returned.

When returning the names of the nodes comprising the node’s path, the names are listed from
the target node to root. For example:
names[0] points to the name of the target node. names[n_name-1] points to the name of the root
node.

RETURN VALUES
KR_ERR_BADKEY The specified key does not exist.
KR_ERR_DELETED  The node represented by the given KRS key has been deleted since the node was opened.

KR_ERR_PARAM  If n_name or buf_sz is NULL, or *buf_sz is not zero and buf is NULL.

KR_ERR_SIZE  The size of names, or buf, or both names and buf, is not enough to fulfill the request.

KR_SUCCESS  Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

int kr_get_node_names(kr_key_t, char *[], int *, char *,
kr_size_t *, kr_flags_t);
int kr_get_value_names(kr_key_t, char *[], int *, char *,
kr_size_t *);
myfunc(kr_key_t key)
{
    /* Error checking and local declarations omitted */
    /* You can use the same call to determine the */
    /* array size or buffer size. */
    rv = kr_get_node_names(key, names, &num_names, buf, &buf_sz,
KR_NOFLAGS);
    if (rv == KR_ERR_SIZE) { /* Array or buffer size is insufficient */
        names = malloc(sizeof(char *) *num_names);
        buf = malloc(buf_sz);
        /* You can use the same call to get the node names. */
        rv = kr_get_node_names(key, names, &num_names, buf, &buf_sz,
KR_NOFLAGS);
    }
    for (i = 0; i < num_names; i++) {
        kr_open_node(key, names[i], KR_NOFLAGS, &ckey);
        ...
        myfunc(ckey);
        kr_close_node(ckey);
    }
}

SEE ALSO

kr_get_node_info(9F), kr_open_node(9F), kr_set_node_flags(9F).
**NAME**

`kr_get_value` - Get the specified named value from an open node.

**SYNOPSIS**

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

int kr_get_value(
    kr_key_t key,
    char *vname,
    kr_type_t *type_buf,
    kr_size_t *size_buf,
    void *buf,
    kr_flags_t flags
);
```

**PARAMETERS**

- **key**
  The KRS key representing the node from which the value is to be read.

- **vname**
  The name of the value to read.

- **type_buf**
  When called, the expected value type. On return, the actual value type.

- **size_buf**
  When called, the size of the client-supplied buffer, pointed to by `buf`. On return, the actual size of the value, or the required size of the buffer if it is not large enough to fulfill the request.

- **buf**
  A pointer to the client-supplied buffer to which the value's data is to be copied.

- **flags**
  The `flags` parameter values are as follows:
  - **KR_NO_INHERIT**
    If the value to be read is not found directly in the target node, the routine does not check to see if the value is exported from any of the node's ancestors.

  - **KR_HOLD**
    Used when reading values of type `KR_VTYPE_TREEREF`. Specifying this flag causes the hold count for the returned key to be incremented, preventing it from being freed while still in use. When the client is finished using the key, call `kr_release_reference`.

**DESCRIPTION**

The `kr_get_value` routine retrieves a value from an open node. The data of the value is copied into the client-supplied buffer pointed to by `buf`. The `size_buf` parameter points to a client-supplied variable containing the size of the buffer. If this size is not large enough to hold the data, the request will fail and the variable pointed to by `size_buf` will be set to the needed size. Upon successful completion, `size_buf` and `type_buf` are set to the actual size and type of value, respectively.

If the value is not found in the target node and the **KR_NO_INHERIT** flag is not specified, each of the target node’s ancestors is searched for the value. The ancestor nodes are searched from the target node’s parent toward the root node. If a value is found with the name `vname` and it has the **KR_EXPORT** flag set, it is returned to the caller.

The following value types are supported:

- **KR_VTYPE_STRING**
  A NULL terminated string.

- **KR_VTYPE_USER**
  User defined. Stored as uninterpreted data and size.

- **KR_VTYPE_INT32**
  A 32-bit integer.

- **KR_VTYPE_INT64**
  A 64-bit integer.
KR_VTYPE_UINT64  An unsigned 64-bit integer.
KR_VTYPE_TREEREF  Tree reference. Stored as a persistent KRS key.
KR_VTYPE_ADDR    A memory address.

RETURN VALUES

KR_ERR_BADKEY    The specified key does not exist.
KR_ERR_DELETED   The node represented by the given KRS key has been deleted since the
                 node was opened.
KR_ERR_NAME      The length of the specified value name exceeds KR_NAME_LEN.
KR_ERR_NOTFOUND  The specified value could not be found.
KR_ERR_PARAM     If type_buf or size_buf are NULL, or if *size_buf is not zero and buf is NULL.
KR_ERR_SIZE      The value is larger than the size of buf.
KR_SUCCESS       Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

```c
int kr_get_value(kr_key_t, char *, kr_type_t *, kr_size_t *,
                 void *, kr_flags_t);
myfunc()
{
    int32_t value;
    kr_type_t type = KR_VTYPE_INT32;
    kr_size_t size = sizeof(int32_t);
    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey);
    kr_get_value(mykey, "v0", &type, &size, &value,
                 KR_NOFLAGS);
    kr_close_node(mykey);
}
```

SEE ALSO

kr_delete_value(9F), kr_open_node(9F), kr_release_reference(9F), kr_set_value(9F).
kr_get_value_names(9F)

NAME

kr_get_value_names - Retrieve a list of value names for the specified node.

SYNOPSIS

#include <sys/krs.h>

int kr_get_value_names(
    kr_key_t key,
    char *names[],
    int *n_name,
    char *buf,
    kr_size_t *buf_sz
);

PARAMETERS

key The KRS key representing the node whose value names are to be read.
names A client-supplied array through which pointers to the value names are returned.
n_name When called, the number of elements in the names array. On return, the actual number of
    names that have been returned or the required size of the names array if it is not
    large enough to fulfill the request.
buf A client-supplied buffer through which the text of the value names are returned.
    Pointers returned in the names array point into this buffer.
buf_sz When called, the size of the buffer pointed to by buf. On return, the actual number of
    characters returned through buf, or the required size of the buffer if it is not large
    enough to fulfill the request.

DESCRIPTION

The kr_get_value_names routine returns the names of all the values in the given node.
The names are returned through the names and buf parameters. The buf parameter points to a
caller-supplied buffer that will contain the text of the names being returned. The names parameter
points to a caller-supplied array of character pointers that will contain pointers to each name in
buf. The n_name and buf_sz parameters point to caller-supplied storage containing the size of the
names array and buf buffer respectively.

If the storage supplied by the caller is not enough to fulfill the request, n_name and buf_sz are set
to indicate the storage required; the request fails with KR_ERR_SIZE. Upon successful completion,
n_name is set to the actual number of names returned.

RETURN VALUES

KR_ERR_BADKEY The specified key does not exist.
KR_ERR_DELETED The node represented by the given KRS key has been deleted since the
    node was opened.
KR_ERR_PARAM If n_name or buf_sz is NULL, or *buf_sz is not zero and buf is NULL.
KR_ERR_SIZE The size of names, or buf, or both names and buf, is not enough to fulfill
    the request.
KR_SUCCESS Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.
EXAMPLES

```c
int kr_get_node_names(kr_key_t, char *[], int *, char *,
                      kr_size_t *, kr_flags_t);
int kr_get_value_names(kr_key_t, char *[], int *, char *,
                       kr_size_t *);
myfunc(kr_key_t key)
{
    /* Error checking and local declarations omitted */
    kr_get_node_names(key, names, &num_names, buf, &buf_sz,
                       KR_NOFLAGS);
    for (i = 0; i < num_names; i++) {
        kr_open_node(key, names[i], KR_NOFLAGS, &ckey);
        ...
        myfunc(ckey);
        kr_close_node(ckey);
    }
}
```

SEE ALSO

`kr_get_vinfo(9F), kr_open_node(9F), kr_set_value_flags`. 
NAME

kr_get_vinfo - Get information about a given named value.

SYNOPSIS

#include <sys/krs.h>

int kr_get_vinfo(
    kr_key_t key,
    char *vname,
    kr_type_t *type_buf,
    kr_size_t *size_buf,
    kr_flag_field_t *flag_buf
);

PARAMETERS

key The KRS key representing the node containing the value whose information is to be obtained.

vname The name of the value whose information is to be obtained.

type_buf A pointer to client-supplied storage through which the value’s type is returned.

size_buf A pointer to client-supplied storage through which the value’s size is returned.

flag_buf A pointer to client-supplied storage through which the value’s flags are returned.

DESCRIPTION

The kr_get_vinfo routine retrieves information specific to the value in question. If the value vname is found in the node associated with key, the value’s size type and flags are returned to the caller.

RETURN VALUES

KR_ERR_BADKEY The specified key does not exist.

KR_ERR_DELETED The node represented by the given KRS key has been deleted since the node was opened.

KR_ERR_NAME The length of the specified value name exceeds KR_NAME_LEN.

KR_ERR_NOTFOUND The specified value could not be found.

KR_SUCCESS Request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

int kr_get_node_info(kr_key_t, kr_flag_field_t, kr_flag_field_t, 
    kr_flag_field_t, kr_flag_field_t, 
    kr_flag_field_t, kr_link_id_t);

int kr_get_vinfo(kr_key_t, char *, kr_type_t *, kr_size_t *, 
    kr_flag_field_t *);

myfunc() {
    /* Error checking and local declarations omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &key);
    kr_get_node_info(key, &info_buf, &vtype_buf, vclass, 
        vphase, &flag_buf, &lid_buf);
    kr_get_vinfo(key, "v0", &type_buf, &size_buf, &vflag_buf);
kr_close_node(key);
}

SEE ALSO

kr_get_value_names(9F), kr_open_node(9F), kr_set_value_flags(9F).
kr_link_node(9F)

NAME

kr_link_node - Create a new registry node as a link to an existing node.

SYNOPSIS

#include <sys/krs.h>

int kr_link_node(
    kr_key_t skey,
    kr_key_t root_key,
    char *path,
    kr_flags_t flags,
    kr_key_t *new_key
);

PARAMETERS

skey  The source key that represents the existing node to which to link.

root_key  The KRS key representing the parent node for the first node defined in path. Here, path is relative to root_key.

path  A string defining the path to the target node. Node names in the path string are separated by the '/' character. If root_key is NULL, the path is assumed to be absolute and must contain a leading '/' character. If root_key is not NULL, the path is assumed to be relative to root_key and must not contain a leading '/' character.

flags  The flags passed through this parameter govern the action of the link operation and define the characteristics of the newly created node. The valid values are as follows:

KR_ALL  Create intermediate nodes in the path to the target node as needed.

KR_NOWAIT  It is not permissible to sleep while fulfilling the link request. Otherwise, the request could block.

KR_NO_PATH_FLAGS  When KR_ALL is used to create intermediate nodes, this flag forces intermediate nodes to have default flag settings. Otherwise, the flags are set to the value of the created target node.

KR_PERSISTENT  The newly created target node needs to be persistent across system reboot.

KR_USER_RDONLY  The new link, any existing links, and the target node are to be read-only from user-space.

KR_PRUNE  Do not save this node or any of its descendents to persistent storage. This effectively removes the branch in question on next reboot, even if information in the branch is flagged as persistent.

new_key  A KRS key representing the new link node is returned through this parameter.

DESCRIPTION

The kr_link_node routine creates a new node that is a link to an existing node. The data and children of the original node are also recognized as the data and children of the new node. The original node is specified through the skey parameter, which contains the key associated with the node (as returned by a previous call to kr_open_node). The target node is specified through root_key and path (as in kr_open_node).
If the target node already exists, the request fails. If any other node in the path does not exist, the request fails, unless the `KR_ALL` flag was specified.

**RETURN VALUES**

- **KR_ERR_BADKEY**: The specified `skey` or `root_key` does not exist.
- **KR_ERR_DELETED**: The node represented by the given KRS key has been deleted since the node was opened.
- **KR_ERR_EXIST**: The target node already exists.
- **KR_ERR_LOOP**: The link would form a loop in the tree.
- **KR_ERR_NAME**: A name component of the path string exceeds `KR_NAME_LEN`.
- **KR_ERR_NOMEN**: When the `KR_NOWAIT` flag is specified, KRS cannot allocate enough memory to fulfill request.
- **KR_ERR_PARAM**: Invalid parameter. (For example: `new_key = NULL`.)
- **KR_ERR_PATH**: The string pointed to by `path` is not a valid path string.
- **KR_ERR_RDONLY**: An attempt was made from user-space to create the link under a parent node that is read-only.
- **KR_ERR_TOODEEP**: The depth of the target node exceeds `KR_MAX_DEPTH`.
- **KR_SUCCESS**: Request completed successfully.

**CONSTRAINTS**

Before using this routine, you must first acquire the key by making a call to `kr_open_node`.

**EXAMPLES**

```c
int kr_link_node(kr_key_t, kr_key_t, char *, kr_flag_t, kr_key_t);
myfunc()
{
    kr_key_t           skey, key;
    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_CREATE, &skey);
    kr_link_node(skey, KR_NOKEY, "/mynodelink", KR_NOFLAGS,
        &key);
    kr_close_node(key);
    kr_close_node(skey);
}
```

**SEE ALSO**

`kr_close_node(9F), kr_open_node(9F)`. 
kr_open_node(9F)

NAME

kr_open_node - Open a given registry tree node for access.

SYNOPSIS

#include <sys/krs.h>

int kr_open_node(
    kr_key_t root_key,  
    char *path,  
    kr_flags_t flags,  
    kr_key_t *new_key
);

PARAMETERS

root_key The KRS key representing the parent node for the first node defined in path. Here, path is relative to root_key.

path A string defining the path to the target node. Node names in the path string are separated by the '/' character. If root_key is NULL, the path is assumed to be absolute and must contain a leading '/' character. If root_key is not NULL, the path is assumed to be relative to root_key and must not contain a leading '/' character.

flags Specifies the action of the open operation and defines the characteristics of any nodes that are created. Valid values are as follows:

    KR_CREATE Create the node or nodes in question if they do not already exist.

    KR_ALL Create intermediate nodes in the path to the target node as needed. This must be used in conjunction with the KR_CREATE flag.

    KR_NOWAIT It is not permissible to sleep while fulfilling the request. Otherwise, the open request could block.

    KR_NO_PATH_FLAGS When KR_ALL is used to create intermediate nodes, this flag forces intermediate nodes to have default flag settings. Otherwise, the flags are set to the value of the created target node.

    KR_PERSISTENT Nodes created will be persistent across system reboot.

    KR_USER_RDONLY Any created nodes will be read only when accessed from user-level applications.

    KR_PRUNE Do not save this node or any of its descendents to persistent storage. This will effectively remove the branch in question on next reboot, even if information in the branch is flagged as persistent.

new_key A KRS key representing the newly opened node is returned through this parameter.

DESCRIPTION

The kr_open_node routine opens a specific node within the KRS tree, returning a key through which the node’s data can be accessed.

The node is identified by path, which can be absolute (starting from root), or relative to root_key (a key returned by a previous call to kr_open_node). The component ".." (dot dot) can be included in a path string to specify the parent of a given node.
If the target node does not exist, it is created if the KR_CREATE flag was specified. If any nodes other than the target node do not exist, they are created if both the KR_CREATE and KR_ALL flags were specified.

**RETURN VALUES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KR_ERR_BADKEY</td>
<td>The specified root_key does not exist.</td>
</tr>
<tr>
<td>KR_ERR_DELETED</td>
<td>The node represented by the given KRS key has been deleted since the node was opened.</td>
</tr>
<tr>
<td>KR_ERR_NAME</td>
<td>A name component of the path string exceeds KR_NAME_LEN.</td>
</tr>
<tr>
<td>KR_ERR_NOMEN</td>
<td>When the KR_NOWAIT flag is specified KRS cannot allocate enough memory to fulfill request.</td>
</tr>
<tr>
<td>KR_ERR_NOTFOUND</td>
<td>The target node does not exist and the KR_CREATE flag has not been specified.</td>
</tr>
<tr>
<td>KR_ERR_PARAM</td>
<td>Invalid parameter. For example: new_key = NULL.</td>
</tr>
<tr>
<td>KR_ERR_PATH</td>
<td>The string pointed to by path is not a valid path string.</td>
</tr>
<tr>
<td>KR_ERR_RDONLY</td>
<td>An attempt was made from user-space to create a node under a parent that is read-only.</td>
</tr>
<tr>
<td>KR_ERR_TOODEEP</td>
<td>The depth of the target node exceeds KR_MAX_DEPTH.</td>
</tr>
<tr>
<td>KR_SUCCESS</td>
<td>Request completed successfully.</td>
</tr>
</tbody>
</table>

**CONSTRAINTS**

Before using this routine, you must first acquire the key by making a call to kr_open_node.

**EXAMPLES**

```c
int kr_open_node(kr_key_t, char *, kr_flags_t, kr_key_t *);  
int kr_close_node(kr_key_t);  
myfunc()  
{  
  int rv;  
  kr_key_t mykey;  
  rv = kr_open_node(KR_NOKEY, "/mynode", KR_CREATE, &mykey);  
  if (rv != KR_SUCCESS) {  
    return ERROR;  
  }  
  ...  
  kr_close_node(mykey);  
}
```

**SEE ALSO**

kr_close_node(9F), kr_delete_node(9F), kr_link_node(9F), kr_release_reference(9F).
kr_release_reference(9F)

NAME

kr_release_reference - Release a hold placed on a tree reference key by kr_get_value.

SYNOPSIS

#include <sys/krs.h>

int kr_release_reference(
    kr_key_t key
);

PARAMETERS

key  The key to be released. This key must be a tree reference obtained through kr_get_value specifying the KR_HOLD flag.

DESCRIPTION

The kr_release_reference routine releases the hold that was placed on a tree reference key that was obtained through a call to kr_get_value. The key passed to this routine must be a tree reference obtained through kr_get_value specifying the KR_HOLD flag. Call this routine when you no longer need the key in question. Once released, do not be use the key without first reacquiring and holding it through kr_get_value.

RETURN VALUES

KR_ERR_BADKEY Indicates that the key in question does not exist, or that it was never held.
KR_SUCCESS The request completed successfully.

EXAMPLES

int kr_release_reference(kr_key_t);

myfunc()
{
    kr_key_t   key, ref_key;
    kr_type_t  type = KR_VTYPE_TREEREF;
    kr_size_t  size = sizeof(kr_key_t);

    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &key);
    kr_get_value(key, "ref", &type, &size, &ref_key, KR_HOLD);
    kr_close_node(key);
    ... /* Use ref_key */
    kr_release_reference(ref_key);
}

SEE ALSO

kr_delete_node(9F), kr_get_value(9F), kr_link_node(9F), kr_open_node(9F).
NAME

kr_set_node_flags - Modify the set of flags associated with a given node.

SYNOPSIS

#include <sys/krs.h>

int kr_set_node_flags(
    kr_key_t key,
    kr_flags_t flags,
    kr_flags_field_t mode
);

PARAMETERS

key A KRS key representing the node whose flags are to be modified.
flags The flag values to be modified according to mode. Valid values are as follows:
   KR_LINKS Apply the flags to the node in question and all nodes that are linked to it.
   KR_PERSISTENT Nodes created will be persistent across system reboot.
   KR_USER_RDONLY Any created nodes will be read-only when accessed from user-level applications.
   KR_PRUNE Do not save this node or any of its descendents to persistent storage. This effectively removes the branch in question on next reboot, even if information in the branch is flagged as persistent.
mode If mode is KR_SET, the flag values in flags are set in the node. If mode is KR_CLEAR, the flag values in flags are cleared.

DESCRIPTION

Use the kr_set_node_flags routine to set or clear attribute flags in an existing node. If mode is KR_SET, the flags specified in flags are set in the node. Otherwise, they are cleared.

RETURN VALUES

KR_ERR_BADKEY The specified key does not exist.
KR_ERR_PARAM The value of mode is not KR_SET or KR_CLEAR; or an attempt was made to set the KR_PRUNE flag on the root node.
KR_ERR_RDONLY An attempt was made from user-level to set the flags of a read-only node.
KR_SUCCESS The request completed successfully.

CONSTRAINTS

Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES

int kr_set_node_flags(kr_key_t, kr_flags_t, kr_flag_field_t);
int kr_set_value_flags(kr_key_t, char *, kr_flags_t , kr_flag_field_t);
myfunc()
{
    kr_key_t mykey1, mykey2;
    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey1);
    kr_open_node(mykey1, "mychild", KR_PRUNE, KR_NOFLAGS, &mykey2);
    int kr_set_node_flags(9F);
    kr_set_node_flags(key, flags, mode);
    int kr_set_value_flags(key, config, flags, field);
}
KR_USER_RDONLY | KR_CREATE, &mykey2);
kr_set_node_flags(mkey1, KR_PERSISTENT1, KR_SET);
kr_set_value_flags(mkey1, "v0", KR_EXPORT, KR_CLEAR);
kr_close_node(mkey1);
kr_close_node(mkey2);
}

SEE ALSO

kr_get_node_info(9F), kr_get_node_names(9F), kr_open_node(9F).
NAME

kr_set_value - Set the specified named value in an open node.

SYNOPSIS

#include <sys/krs.h>

int kr_set_value(
    kr_key_t key,
    char *vname,
    kr_type_t type,
    kr_size_t size,
    void *buf,
    kr_flags_t flags
);

PARAMETERS

key The KRS key representing the node in which the value is to be set.
vname The name of the value being set.
type The type of the value being set.
size The size of the value being set.
buf A pointer to the value's data.
flags Flags governing the action of the set value operation and defining characteristics of
the value being set. Valid values are as follows:

KR_NOWAIT It is not permissible to sleep while fulfilling the set value request. Otherwise, the request might block.

KR_PERSISTENT The value being set needs to be persistent across system reboot.

KR_USRBUF The data for the value being set is to be maintained in the buffer supplied by the client. Otherwise, a new buffer will be allocated by KRS.

KR_EXPORT The value being set is to be visible to all lower nodes in the branch.

KR_USER_RDONLY Any created values will be read-only when accessed from user-level applications.

DESCRIPTION

The kr_set_value routine creates or changes a value in the specified node. The data for the
value in question is specified through buf, a pointer to storage containing the data in question.
Normally, KRS allocates storage to hold the data for the value, and the data is copied from the
user's buffer to the newly allocated storage. If the KR_USRBUF flag is specified however, the
buffer supplied by the caller is used instead. It is the caller's responsibility to ensure that the
buffer exists for the lifetime of the value.

The following value types are supported:

KR_VTYPE_STRING A NULL terminated string.
KR_VTYPE_USER User-defined. Stored as un-interpreted data and size.
KR_VTYPE_INT64 A 64-bit integer.
KR_VTYPE_UINT64  An unsigned 64-bit integer.
KR_VTYPE_TREEREF  Tree reference. Stored as a persistent KRS key.
KR_VTYPE_ADDR    A memory address.

RETURN VALUES
KR_ERR_BADKEY    The specified key does not exist.
KR_ERR_DELETED   The node represented by the given KRS key has been deleted since
                 the node was opened.
KR_ERR_INVAL_FLAG Invalid flag specification.
KR_ERR_NAME      The length of the specified value name exceeds KR_NAME_LEN.
KR_ERR_NOMEN     When the KR_NOWAIT flag is specified, KRS cannot allocate enough
                 memory to fulfill request.
KR_ERR_PARAM     Invalid parameter/flag combination.
KR_ERR_RDONLY    An attempt was made, from user-space, to change a read-only value,
                 or change or create a value in a read-only node.
KR_ERR_TYPE      Unrecognized type specified.
KR_SUCCESS       Request completed successfully.

CONSTRAINTS
Before using this routine, you must first acquire the key by making a call to kr_open_node.

EXAMPLES
int kr_set_value(kr_key_t, char *, kr_type_t, kr_size_t, void *,
                 kr_flags_t);
myfunc()
{
    int32_t value = 123;
    kr_type_t type = KR_VTYPE_INT32;
    kr_size_t size = 0;

    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey);
    kr_set_value(myKey, "v0", type, size, &value, KR_NOFLAGS);
    kr_close_node(mykey);
}

SEE ALSO
kr_delete_value(9F), kr_get_value(9F), kr_open_node(9F).
**NAME**

kr_set_value_flags - Modify the set of flags associated with a given value.

**SYNOPSIS**

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

int kr_set_value_flags(
    kr_key_t key,
    char *vname,
    kr_flags_t flags,
    kr_flag_field_t mode
);
```

**PARAMETERS**

- **key**  
  A KRS key representing the node containing the value whose flags are to be modified.

- **vname**  
  The name of the value whose flags are to be modified.

- **flags**  
  The flag values to be modified according to `mode`. Valid values are as follows:
  - **KR_PERSISTENT**  
    The value to be set needs to be persistent across system reboot.
  - **KR_USRBUF**  
    The data for the value to be set is to be maintained in the buffer supplied by the client. Otherwise, KRS allocates a new buffer.
  - **KR_EXPORT**  
    The value being set is to be visible to all lower nodes in the branch.
  - **KR_USER_RDONLY**  
    Any created nodes will be read-only when accessed from user-level applications.

- **mode**  
  Determines how the flag values are modified. Valid `mode` values are as follows:
  - **KR_SET**  
    The flag values in `flags` are set in the value.
  - **KR_CLEAR**  
    The flag values in `flags` are cleared in the non-pending value.

**DESCRIPTION**

Use the `kr_set_value_flags` routine to set or clear attribute flags for an existing value. If `mode` is **KR_SET**, the flags specified in `flags` are set in the node. Otherwise, they are cleared.

**RETURN VALUES**

- **KR_ERR_BADKEY**  
  The specified `key` does not exist.

- **KR_ERR_NAME**  
  The length of the specified value name exceeds **KR_NAME_LEN**.

- **KR_ERR_NOTFOUND**  
  The specified value could not be found.

- **KR_ERR_PARAM**  
  The value of `mode` is not **KR_SET** or **KR_CLEAR**.

- **KR_ERR_RDONLY**  
  An attempt was made, from user-space, to set the flags of a read-only value or to set the flags of a value in a read-only node.

- **KR_SUCCESS**  
  The request completed successfully.

**CONSTRAINTS**

Before using this routine, you must first acquire the key by making a call to `kr_open_node`.

**EXAMPLES**

```c
int kr_set_node_flags(kr_key_t, kr_flags_t, kr_flag_field_t);
int kr_set_value_flags(kr_key_t, char *, kr_flags_t, kr_flag_field_t);
```
myfunc()
{
    kr_key_t mykey1, mykey2;

    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey1);
    kr_open_node(mykey1, "mychild",
                  KR_USER_RDONLY | KR_CREATE, &mykey2);
    kr_set_node_flags(mykey1, KR_PERSISTENT1, KR_SET);
    kr_set_value_flags(mykey1, "v0", KR_EXPORT, KR_CLEAR);
    kr_close_node(mykey1);
    kr_close_node(mykey2);
}

SEE ALSO

kr_get_value_names(9F), kr_get_vinfo(9F), kr_open_node(9F).
NAME

Ktimeout - Execute a callout function after a specified length of time at driver level interrupt priority.

SYNOPSIS

#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 to untimetypeout.

RETURN VALUES

The Ktimeout function returns a pointer to a callout structure.

CONSTRAINTS

Do not call 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.

EXAMPLE

/*
 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(9F), untimeout(9F)
NAME

ktune_canauto - Checks whether a tunable is capable of being automatically tuned.

SYNOPSIS

#include <sys/ktune.h>

int ktune_canauto(
    ktune_id_t tuneid
);

PARAMETERS

tuneid    ID of the tunable to check.

DESCRIPTION

This function checks whether the tunable with the specified ID is capable of being automatically tuned. A tunable is considered to be capable of being automatically tuned if it has one or more registered handlers capable of computing an algorithmic default value, and has one or more registered handlers capable of making dynamic changes to the tunable’s value.

RETURN VALUES

<>0    Tunable is capable of being automatically tuned.

0      Tunable is not capable of being automatically tuned.

CONSTRAINTS

Do not call while holding a spinlock with order >= KTUNE_LISTLOCK_ORDER.

EXAMPLES

if (ktune_canauto(tuneid)) {
    ...
}

SEE ALSO

ktune_id(9F), ktune_isauto(9F)
ktune_current(9F)

NAME

ktune_current - Get the current value of a tunable from a tunable handler.

SYNOPSIS

#include <sys/ktune.h>

int ktune_current(
    ktune_id_t tuneid,
    uint64_t *value
);

PARAMETERS

tuneid     ID of the tunable of interest.
value      Pointer to a variable into which the current value of the tunable will be placed.

DESCRIPTION

This function retrieves the current value of a tunable. It is for use within tunable handlers only. Other callers wanting the current value of a tunable must call ktune_get.

The value returned by this function ignores any tunable change transaction which might be in progress. To retrieve the pending value of a tunable change, call ktune_pending.

RETURN VALUES

0     Success
<>0   Error code returned from tunable handler

CONSTRAINTS

Call this function only from a tunable handler.

EXAMPLES

/* From within a tunable handler: */
ret = ktune_current(event_data->kte_tuneid, &currentval);
if (ret)
    ...

SEE ALSO

ktune_event_t(9F), ktune_get(9F), ktune_handler(9F), ktune_id(9F), ktune_pending(9F).
ktune_error(9F)

NAME

ktune_error - Log a tunable error message from a tunable handler.

SYNOPSIS

#include <sys/ktune.h>

void ktune_error(
    ktune_txn_id_t txnid,
    const char *format,
    ...
);

PARAMETERS

txin
    ID of the transaction in flight, if any.
format
    printf-style format string.
... 
    Arguments needed for format string.

DESCRIPTION

This function logs a tunable change error message. If a tunable change transaction is in progress, the error message will be returned to the caller of settune. Otherwise, it will be logged to the system message buffer. This function must only be called from a tunable handler.

Do not embed tunable names in error message strings. Instead, put a %s where the name is to go, and use ktune_name as an argument to fill in the name.

RETURN VALUES

None

CONSTRAINTS

Call this function only from a tunable handler.

EXAMPLES

/* From within a tunable handler: */
ktune_error(event_data->kte_txnid,
    "%lu is an invalid value for %s."
    ,
    value, ktune_name(event_data->kte_tuneid));

SEE ALSO

ktune_event_t(9F), ktune_handler(9F), ktune_id(9F), ktune_name(9F), ktune_warning(9F).
NAME

ktune_event_t - Details of a tunable handler request.

SYNOPSIS

#include <sys/ktune.h>

typedef struct ktune_event {
    ktune_id_t kte_tuneid;
    ktune_txnid_t kte_txnid;
    ktune_op_t kte_op;
    ktune_event_flags_t kte_flags;
    ktune_id_t kte_baseid;
} ktune_event_t;

DESCRIPTION

A pointer to a ktune_event_t structure is passed as the event_data parameter in any invocation of a tunable handler that has a reason code of KEN_EVENT or KEN_BACKOUT. The data in this structure gives information about what the handler is expected to do.

STRUCTURE MEMBERS

kte_tuneid  Tunable ID of the tunable for which the handler is being called.
kte_txnid  Contains the transaction ID of the tunable change transaction in progress, if any. If no transaction is in progress, this value will be KTUNE_TXN_NULL.
kte_op  Specifies the operation that the handler is supposed to perform. Valid values are KTOP_CAPABLE, KTOP_GETDEFAULT, KTOP_VALIDATE, KTOP_PREPARE, KTOP_COMMIT, and KTOP_NOTIFY.
kte_flags  Additional information about the tunable and the transaction. Valid flags are:
            KTF_DEFAULT  Specifies that the tunable is in its default state.
            KTF_NEXTBOOT  Specifies that the transaction in progress will not take effect until boot.
kte_baseid  If the tunable for which the handler is being called is a derived tunable, this field contains the tunable ID of the corresponding base tunable; otherwise, this field contains KTUNE_ID_NULL.

SEE ALSO

ktune_handler(9F).
NAME
ktune_get - Activates a tunable and returns its value.

SYNOPSIS
#include <sys/ktune.h>

uint64_t ktune_get(
    const char *tunable,
    uint64_t failsafe
);

PARAMETERS
tunable Name of the tunable to activate.
failsafe Value to use for the tunable if stored values are unavailable.

DESCRIPTION
This function activates a tunable. Tunables are inactive at boot time until this function is first called.
Before activating a tunable, all handlers for that tunable must be registered using ktune_register_handler.
If tunable failsafe mode was engaged at boot time (using the -t m boot flag), or if an error occurs trying to retrieve tunable values from the kernel registry, this function will return the failsafe value passed into it.
If the tunable is not in its default state, the saved value of the tunable is validated. If the saved value is no longer valid, a warning message is printed and the tunable is put into its default state.
If the tunable is in its default state, and a handler is registered that can compute default values, a new default value is calculated.
The resulting value of the tunable is returned.
This function may be used when the tunable is already active. In such cases it merely returns the current value of the tunable.

RETURN VALUES
Value of the tunable if it is found and the kernel is not in failsafe mode.
Failsafe value that is passed in otherwise.

CONSTRAINTS
Do not call while holding a spinlock.
Do not call in interrupt context.

EXAMPLE
maxuprc = ktune_get(“maxuprc”, MAXUPRC_FAILSAFE);

SEE ALSO
ktune_register_handler(9F).
ktune_handler(9F)

NAME

ktune_handler - Tunable handler function.

SYNOPSIS

#include <sys/ktune.h>

int ktune_handler(
    ken_id_t eventid,
    int reason,
    ken_instance_t instance,
    void *handler_data,
    ktune_event_t *event_data,
    int *result
);

PARAMETERS

eventid    ID of the KEN event for the tunable.
reason     Reason the handler was invoked, one of:
            KEN_REGISTER    The handler is being registered.
            KEN_EVENT      The handler needs to perform a tunable operation.
            KEN_BACKOUT    The handler needs to reverse a tunable operation.
            KEN_UNREGISTER The handler is being unregistered.
instance   ID of this particular invocation of this handler.
handler_data A copy of the handler_data parameter provided to
              ktune_register_handler when the handler was registered.
event_data When the handler is invoked with reason set to KEN_EVENT or KEN_BACKOUT,
              this parameter points to a ktune_event_t structure describing the tunable
              operation to be performed or reversed. For other reason values, this parameter
              is NULL.
result     Points to an integer variable into which the handler must store its result code.

DESCRIPTION

A tunable handler is a function provided by a kernel subsystem that embodies knowledge of
how to manage kernel tunables owned by that subsystem.

A tunable handler can be registered for multiple tunables. Multiple tunable handlers can be
registered for a tunable. A tunable handler may be registered more than once for the same tunable
as long as the associated handler_data parameters differ. Tunable handlers are registered using
the ktune_register_handler call.

The actions taken by a handler depend on the reason and operation codes it receives. When the
reason code is KEN_EVENT or KEN_BACKOUT, the event_data parameter will point to a
ktune_event_t structure, which contains an operation code in its kte_op field (see
ktune_event_t(9F).) Valid operation codes are:

KTOP_CAPABLE    Return the capabilities of the handler.
KTOP_COMMIT     Commit the change to the value of the tunable.
KTOP_GETDEFAULT Compute a default value for the tunable.
KTOP_NOTIFY     Adjust to a new value of the tunable.
KTOP_PREPARE    Prepare to change the value of the tunable.
KTOP_VALIDATE Validate a proposed value for the tunable.

All handlers are required to respond to a reason/operation code combination of KEN_EVENT/KTOP_CAPABLE by setting their outbound result parameter to the bitwise OR of the KTOP_* constants for the operations implemented by the handler. Handling of all other reason and operation codes is optional.

When responding to a request, handlers take appropriate action for the request, set their outbound result parameter to a valid result code for the request, and return KEN_DONE. When handlers receive a request they do not implement, they must take no action other than returning KEN_DONE.

There are a variety of functions provided by the tunable infrastructure for use by handlers. See the SEE ALSO section.

RETURN VALUES

KEN_DONE Handler functions always return KEN_DONE.

CONSTRAINTS

Handler functions may be called during early boot or during a system call. They will not be called from the interrupt stack. They will not be called with any locks held, and must not hold any locks when they return.

Handlers that wish to sleep must first verify that the system has finished booting. Aside from that restriction, handlers may sleep.

EXAMPLES

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

int template_tunable_handler(ken_id_t eventid,
          int reason,
          ken_instance_t instance,
          void *handler_data,
          ktune_event_t *event_data,
          int *result)
{
    switch (reason) {
        case KEN_EVENT: {
            switch (event_data->kte_op) {
                case KTOP_CAPABLE:
                    *result = KTOP_CAPABLE | KTOP_... | ...;
                    break;
                case KTOP_VALIDATE:
                    ...
                    break;
                case KTOP_GETDEFAULT:
                    ...
                    break;
                case KTOP_PREPARE:
                    ...
                    break;
                case KTOP_COMMIT:
                    ...
                    break;
                case KTOP_NOTIFY:
                    ...
                    break;
                default:
                    break;
            }
        }
        break;
```
case KEN_BACKOUT:
    switch (event_data->kte_op) {
    case KTOP_PREPARE:
        ...
        break;
    default:
        break;
    }
    break;

default:
    break;
}

return KEN_DONE;

SEE ALSO
ktune_canauto(9F), ktune_current(9F), ktune_error(9F), ktune_event_t(9F), ktune_id(9F), ,
ktune_isauto(9F), ktune_isdefault(9F), ktune_name(9F), ktune_pending(9F), ktune_register_handler(9F),
ktune_savedefault(9F), ktune_simple_constraint(9F), ktune_simple_dynamic(9F),
ktune_unregister_handler(9F), ktune_validate_powerof2(9F), ktune_validate_zero_or_min(9F),
ktune_warning(9F).

NOTES
Pre-written tunable handlers are available for common cases. The ktune_validate_powerof2
function validates that a proposed value of a tunable is a power of two. The
ktune_validate_zero_or_min function validates that a proposed value of a tunable is either
zero or greater than some minimum value specified when the handler is registered. The
ktune_simple_constraint function validates that the relationship between two tunable
values meets a simple constraining equation specified when the handler is registered. The
ktune_simple_dynamic function implements dynamic behavior for an integer tunable that
requires no locking. For details on each, see their respective manpages.
NAME

ktune_id -- Get the ID of a tunable.

SYNOPSIS

#include <sys/ktune.h>

ktune_id_t ktune_id(
    const char *tunable
);

PARAMETERS

tunable Name of the tunable.

DESCRIPTION

This function returns the tunable ID for the tunable with the given name.

RETURN VALUES

KTUNE_ID_NULL No tunable was found with the specified name.

CONSTRAINTS

Do not call while holding a spinlock with order >= KTUNE_LISTLOCK_ORDER.

EXAMPLES

ktune_id_t tuneid = ktune_id("maxuprc");

SEE ALSO

ktune_get(9F), ktune_name(9F).
NAME

ktune_inactive - Marks a tunable as no longer in use.

SYNOPSIS

#include <sys/ktune.h>

void ktune_inactive(
        ktune_id_t tuneid
    );

PARAMETERS

tuneid ID of the tunable to mark inactive.

DESCRIPTION

This function tells the tunable infrastructure that the tunable with the specified ID is no longer
in use by kernel code. Typically this is only called by a DLKM during the module unload process.
The tunable ID can be obtained by calling ktune_id.

This function does not remove a tunable from the list of tunables seen by an administrator.
However, it is marked as inactive in such lists.

The tunable infrastructure will allow, and apply immediately, any change to an inactive tunable
that is consistent with the tunable's min and max boundaries, if any. No other validation is
performed until the tunable is next activated.

All tunables are considered inactive at boot time until they are activated. Tunables are activated
by calling ktune_get.

No handlers may be registered for a tunable when it is made inactive.

RETURN VALUES

None

CONSTRAINTS

Do not call while holding a spinlock with order >= KTUNE_LISTLOCK_ORDER.

EXAMPLES

ktune_inactive(tuneid);

SEE ALSO

ktune_get(9F), ktune_id(9F).
ktune_isauto(9F)

NAME

ktune_isauto - Checks whether a tunable is being automatically tuned.

SYNOPSIS

#include <sys/ktune.h>

int ktune_isauto(
    ktune_id_t tuneid
);

PARAMETERS

tuneid    ID of the tunable to check.

DESCRIPTION

This function checks whether the tunable with the specified ID is being automatically tuned. A
 tunable is considered to be automatically tuned if it is in its default state (see ktune_isdefault(9F)),
 has one or more registered handlers capable of computing an algorithmic default value, and has
 one or more registered handlers capable of making dynamic changes to the tunable’s value.

RETURN VALUES

<>0    Tunable is being automatically tuned.
 0    Tunable is not being automatically tuned.

CONSTRAINTS

Do not call while holding a spinlock with order \geq KTUNE_LISTLOCK_ORDER.

EXAMPLES

if (ktune_isauto(tuneid)) {
    ...
}

SEE ALSO

ktune_canauto(9F) ktune_id(9F), ktune_isdefault(9F).
NAME

ktune_isdefault - Checks whether a tunable is in its default state.

SYNOPSIS

#include <sys/ktune.h>

int ktune_isdefault(
    ktune_id_t tuneid
);  

PARAMETERS

tuneid  ID of the tunable to check.

DESCRIPTION

This function checks whether the tunable with the specified ID is in its default state. Note that
the default state for the tunable may be a fixed default value, or it may be a self-tuning algorithm;
you cannot tell by using this function. (To find out whether a tunable is self-tuning, use
ktune_isauto.)

RETURN VALUES

<>0   Tunable is in its default state.
0     Tunable is not in its default state.

CONSTRAINTS

Do not call while holding a spinlock with order ≥ KTUNE_LISTLOCK_ORDER.

EXAMPLES

if (ktune_isdefault(tuneid)) {
    ...
}  

SEE ALSO

ktune_id(9F), ktune_isauto(9F).  

NAME

ktune_isdynamic - Checks whether a tunable can be tuned without a reboot.

SYNOPSIS

#include <sys/ktune.h>

int ktune_isdynamic(
    ktune_id_t tuneid
);

PARAMETERS

tuneid    ID of the tunable to check.

DESCRIPTION

This function checks whether the tunable with the specified ID can be tuned without a reboot. A tunable is considered to be dynamically tuned if it has a registered handler capable of making dynamic changes to the tunable's value (The handler supports KTOP_COMMIT. See ktune_handler(9F) for information).

RETURN VALUES

<>0  Tunable changes take effect immediately.
0   Tunable changes take effect at next boot.

CONSTRAINTS

Do not call while holding a spinlock with order \( \geq \) KTUNE_LISTLOCK_ORDER.

EXAMPLES

if (ktune_isdynamic(tuneid)) {
    ...
}

SEE ALSO

ktune_id(9F), ktune_handler(9F).
ktune_name(9F)

NAME

ktune_name - Get the name(s) of a tunable.

SYNOPSIS

#include <sys/ktune.h>
const char *ktune_name(
    ktune_id_t tuneid
);

PARAMETERS

tuneid ID of the tunable whose name is needed.

DESCRIPTION

This function returns a pointer to a character string identifying the tunable with the given ID. This string is appropriate to use in error and warning messages. (See ktune_error(9F) and ktune_warning(9F).)

This function is used to make the handler functions generic to be used for many tunables at the same time.

The caller must not modify the string.

RETURN VALUES

Pointer to the string containing the tunable name.

CONSTRAINTS

This function must only be called from a tunable handler.

EXAMPLES

ktune_error(KTUNE_TXN_NULL,
            "%s has a bad value",
            ktune_name(tuneid));

SEE ALSO

ktune_error(9F), ktune_id(9F), ktune_warning(9F).
NAME

ktune_pending - Get the pending value of a tunable.

SYNOPSIS

#include <sys/ktune.h>

int ktune_pending(
    ktune_txn_id_t txnid,
    ktune_id_t tuneid,
    uint64_t *value
);

PARAMETERS

txnid ID of the transaction in flight, if any.
tuneid ID of the tunable of interest.
value Pointer to a variable into which the pending value of the tunable will be placed.

DESCRIPTION

This function retrieves the pending value of a tunable: the value that the tunable will have if the
specified tunable change transaction completes successfully. More specifically:

1. If the transaction involves changing this tunable to a specific value, that value will be
   returned.
2. If the transaction will be applied to take effect at next boot, then the value that will be used
   at next boot will be returned.
3. If the transaction will be applied to the running kernel and does not involve the tunable at
   all, its current value will be returned.
4. If the transaction involves setting this tunable to its default state, the default value will be
   returned.
5. If no transaction is specified (the txnid parameter is KTUNE_TXN_NULL), the current value
   will be returned.

This function is for use within tunable handlers only.

To retrieve the current value of a tunable, regardless of any transaction in flight, call
ktune_current.

RETURN VALUES

0 Success
&lt;0 Error code returned from tunable handler.

CONSTRAINTS

This function must only be called from a tunable handler.

EXAMPLES

/* From within a tunable handler: */
ret = ktune_pending(event_data-&gt;kte_txnid,
    event_data-&gt;kte_tuneid,
    &pendingval);
if (ret)
    ...

134 Kernel Reference Pages
SEE ALSO

ktune_current(9F), ktune_event_t(9F), ktune_get(9F), ktune_handler(9F), ktune_id(9F).
NAME

ktune_register_handler - Register a handler function for a kernel tunable.

SYNOPSIS

#include <sys/ktune.h>

int ktune_register_handler
    (int version,
     const char *tunable,
     int ken_flags,
     int ken_order,
     const char *description,
     ktune_handler_t handler,
     void *handler_data)
;

PARAMETERS

version        Must be set to KTUNE_VERSION.
tunable        Name of the tunable for which to register the handler.
ken_flags      Must be set to zero.
ken_order      Must be set to KEN_UNORDERED.
description    Description of the handler (for debugging purposes).
handler        Function pointer of the handler.
handler_data   Data to be passed to the handler each time it is called.

DESCRIPTION

This function registers a tunable handler for the specified tunable. For details on tunable handlers, see ktune_handler(9F).

All tunable handlers must be registered before calling ktune_get to activate the tunable.

RETURN VALUES

0         success
-1         invalid data
everything else  a KEN_* error code

CONSTRAINTS

May be called in user context or during boot.
Do not call in interrupt context.
Do not call while holding a spinlock.

EXAMPLES

/* Register a power-of-two validation handler for
 * the tunable vas_hash_locks.
 */
ret = ktune_register_handler(KTUNE_VERSION, "vas_hash_locks",
    0, KEN_UNORDERED,
    "vas_hash_locks validate power of 2",
    (ktune_handler_t)ktune_validate_powerof2,
    NULL);
WARNINGS

The handler will be called twice before this function returns. It will be called with a reason code of KEN_EVENT and an operation code of KTOP_CAPABLE. It will also be called with a reason code of KEN_REGISTER.

SEE ALSO

ktune_get(9F), ktune_handler(9F), ktune_unregister_handler(9F).
NAME

ktune_savedefault - Save a new default value for a tunable.

SYNOPSIS

#include <sys/ktune.h>
void ktune_savedefault(
    ktune_txn_id_t txnid,
    ktune_id_t tuneid,
    uint64_t defvalue
);

PARAMETERS

txnid  ID of the transaction in progress, if any; otherwise, KTUNE_TXN_NULL.
tuneid  ID of the tunable that has a new default value.
defvalue  new default value for the tunable.

DESCRIPTION

This function saves a new default value for a tunable. This function would be called by a tunable
handler that was asked to compute a new default value (a KTOP_GETDEFAULT request). The
txnid and tuneid parameters must be those passed to the handler in its ktune_event_t structure.

RETURN VALUES

None

CONSTRAINTS

This function must only be called from a tunable handler.

EXAMPLES

/* in a tunable handler */
case KTOP_GETDEFAULT:
    ktune_savedefault(event_data->kte_txnid, 
        event_data->kte_tuneid, 
        new_default_value);
    break;

SEE ALSO

ktune_event_t(9F), ktune_handler(9F), ktune_id(9F).
ktune_simple_constraint(9F)

NAME

ktune_simple_constraint - Handler to validate tunables related by a constraint.

SYNOPSIS

#include <sys/ktune.h>

int ktune_simple_constraint(
    ken_id_t eventid,
    int reason,
    ken_instance_t instance,
    ktune_simple_constraint_data_t *constraint,
    ktune_event_t *event_data,
    int *result
);

PARAMETERS

eventid The ID of the event.
reason Reason the handler was invoked:
KEN_REGISTER The handler is being registered.
KEN_EVENT The handler needs to perform a tunable operation.
KEN_BACKOUT The handler needs to reverse a tunable operation.
KEN_UNREGISTER The handler is being unregistered.

instance ID of this particular invocation of this handler.
constraint Pointer to the ktune_simple_constraint_data_t structure which defines
the constraint between the 2 tunables.

event_data When the handler is invoked with reason set to KEN_EVENT or KEN_BACKOUT,
this parameter points to a ktune_event_t structure describing the tunable
operation to be performed or reversed. For other reason values, this parameter
is NULL.

result Points to an integer variable into which the handler must store its result code.

DESCRIPTION

This is a generic tunable handler that validates any constraint between two tunables that can be
written in the form: (scale_a * tunable_a) <= (scale_b * tunable_b) + offset.
This handler requires that the tunables are of integer type. This tunable handler supports the
KTOP_VALIDATE operation.

While registering the handler, the address of the ktune_simple_constraint_data_t structure
describing the constraint must be supplied as the handler_data parameter to
ktune_register_handler. If the result parameter points to the value zero after the handler
is called, the values for the tunables satisfy the constraint. Note that this handler must not be
explicitly registered if the constraint is specified in the modmeta file for the module defining the
tunables.

RETURN VALUES

KEN_DONE Handler functions always return KEN_DONE.

CONSTRAINTS

See ktune_handler(9F) for information on constraints for tunable handlers.
EXAMPLES

To validate the constraint `tunable_a < tunable_b`:
```
#include <sys/ktune.h>
static ktune_simple_constraint_data_t constraint;
{
    int ret
    ...
    constraint.ksc_name_a = "tunable_a";
    constraint.ksc_name_b = "tunable_b";
    constraint.ksc_scale_a = 1;
    constraint.ksc_scale_b = 1;
    constraint.ksc_offset  = -1;
    ret = ktune_register_handler(KTUNE_VERSION, "tunable_a",
                                  0, KEN_UNORDERED,
                                  "Constraint handler for tunable_a",
                                  (ktune_handler_t)ktune_simple_constraint,
                                  &constraint);

    ret = ktune_register_handler(KTUNE_VERSION, "tunable_b",
                                  0, KEN_UNORDERED,
                                  "Constraint handler for tunable_b",
                                  (ktune_handler_t)ktune_simple_constraint,
                                  &constraint);
    ...
}
```

To validate the constraint `2*tunable_a <= tunable_b`:
```
#include <sys/ktune.h>
static ktune_simple_constraint_data_t constraint;
{
    int ret
    ...
    constraint.ksc_name_a = "tunable_a";
    constraint.ksc_name_b = "tunable_b";
    constraint.ksc_scale_a = 2;
    constraint.ksc_scale_b = 1;
    constraint.ksc_offset  = 0;
    ret = ktune_register_handler(KTUNE_VERSION, "tunable_a",
                                  0, KEN_UNORDERED,
                                  "Constraint handler for tunable_a",
                                  (ktune_handler_t)ktune_simple_constraint,
                                  &constraint);

    ret = ktune_register_handler(KTUNE_VERSION, "tunable_b",
                                  0, KEN_UNORDERED,
                                  "Constraint handler for tunable_b",
                                  (ktune_handler_t)ktune_simple_constraint,
                                  &constraint);
```
SEE ALSO

ktune_handler(9F), ktune_event_t(9F), ktune_simple_constraint_data_t(9F), ktune_register_handler(9F).
NAME

ktune_simple_constraint_data_t - Details for the simple constraint handler.

SYNOPSIS

#include <sys/ktune.h>

typedef struct ktune_simple_constraint_data {
    const char *ksc_name_a;
    const char *ksc_name_b;
    int64_t     ksc_scale_a;
    int64_t     ksc_scale_b;
    int64_t     ksc_offset;
    ktune_id_t  ksc_tuneid_a;
    ktune_id_t  ksc_tuneid_b;
} ktune_simple_constraint_data_t;

DESCRIPTION

The ktune_simple_constraint_data_t structure is used to represent the constraint expression that relates two tunables. A constraint expression can take the following form:

(scale_a * tunable_a) <= (scale_b * tunable_b) + offset

The ktune_simple_constraint_handler tunable handler uses a pointer to the ktune_simple_constraint_data_t structure to validate the constraint expression. Either the tunable name or tunable ID fields in the structure need to be filled in. If the tunable ID fields are not filled in, they must be set to KTUNE_ID_NULL.

STRUCTURE MEMBERS

ksc_name_a The name of the first tunable in the constraint expression.
ksc_name_b The name of the second tunable in the constraint expression.
ksc_scale_a The scaling factor for the first tunable.
ksc_scale_b The scaling factor for the second tunable.
ksc_offset The offset to be applied.
ksc_tuneid_a The tunable ID for the first tunable.
ksc_tuneid_b The tunable ID for the second tunable.

SEE ALSO

ktune_handler(9F), ktune_simple_constraint(9F)
NAME

ktune_simple_dynamic - Allows dynamic changes to an integer tunable that needs no locking.

SYNOPSIS

#include <sys/ktune.h>

int ktune_simple_dynamic(
    ken_id_t eventid,
    int reason,
    ken_instance_t instance,
    int *tune_var,
    ktune_event_t *event_data,
    int *result
);

PARAMETERS

**eventid**  
The ID of the event.

**reason**  
Reason the handler was invoked:
- KEN_REGISTER  
The handler is being registered.
- KEN_EVENT  
The handler needs to perform a tunable operation.
- KEN_BACKOUT  
The handler needs to reverse a tunable operation.
- KEN_UNREGISTER  
The handler is being unregistered.

**instance**  
ID of this particular invocation of this handler.

**tune_var**  
Pointer to the tunable variable.

**event_data**  
When the handler is invoked with **reason** set to KEN_EVENT or KEN_BACKOUT, this parameter points to a ktune_event_t structure describing the tunable operation to be performed or reversed. For other reason values, this parameter is NULL.

**result**  
Points to an integer variable into which the handler must store its result code.

DESCRIPTION

This is a generic tunable handler to make a tunable dynamic. This handler does not use any locking while changing the value of the tunable. This handler must be registered only for tunables that are of integer type. This tunable handler supports the KTOP_COMMIT operation.

While registering the handler, the address of integer variable with the tunable value must be supplied as the handler_data parameter to ktune_register_handler.

RETURN VALUES

KEN_DONE  
Handler functions always return KEN_DONE.

CONSTRAINTS

See ktune_handler(9F) for information on constraints for tunable handlers.

EXAMPLES

#include <sys/ktune.h>

int mytunable;
{
    int ret
ret = ktune_register_handler(KTUNE_VERSION, "mytunable",
0, KEN_UNORDERED,
"Dynamic handler for mytunable",

(ktune_handler_t)ktune_simple_dynamic,

&mymutable);

SEE ALSO

ktune_current(9F), ktune_handler(9F), ktune_event_t(9F), ktune_register_handler(9F).
ktune_unregister_handler(9F)

NAME

ktune_unregister_handler - Unregister a handler function for a kernel tunable.

SYNOPSIS

#include <sys/ktune.h>

int ktune_unregister_handler(
    int version,
    const char *tunable,
    int ken_flags,
    int ken_order,
    ktune_handler_t handler,
    void *handler_data
);

PARAMETERS

version Must be set to KTUNE_VERSION.
tunable Name of the tunable from which to unregister the handler.
ken_flags Must be set to the same value given in the ktune_register_handler call when the handler was registered.
ken_order Must be set to the same value given in the ktune_register_handler call when the handler was registered.
handler Function pointer of the handler to unregister.
handler_data Must be set to the same value given in the ktune_register_handler call when the handler was registered.

DESCRIPTION

This function unregisters a tunable handler for the specified tunable. The tunable handlers registered by a module need to be unregistered by the module unload function.

RETURN VALUES

0 success
-1 invalid data
everything else a KEN_* error code

CONSTRAINTS

May be called in user context or during boot.
Do not call in interrupt context.
Do not call while holding a spinlock.

EXAMPLES

/* Unregister a handler for the tunable vas_hash_locks. */
ret = ktune_unregister_handler(KTUNE_VERSION, "vas_hash_locks",
    0, KEN_UNORDERED,
    (ktune_handler_t)ktune_validate_powerof2,
    NULL);
WARNINGS

The handler will be called twice before this function returns. It will be called with a reason code of KEN_EVENT and an operation code of KTOP_CAPABLE. It will also be called with a reason code of KEN_UNREGISTER.

SEE ALSO

ktune_handler(9F), ktune_inactive(9F), ktune_register_handler(9F).
**NAME**

ktune_validate_powerof2 - Handler to validate that the tunable is a power of 2.

**SYNOPSIS**

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

int ktune_validate_powerof2(
    ken_id_t eventid,
    int reason,
    ken_instance_t instance,
    void *unused,
    ktune_event_t *event_data,
    int *result
);
```

**PARAMETERS**

- **eventid**
  The ID of the event.

- **reason**
  Reason the handler was invoked:
  - KEN_REGISTER: The handler is being registered.
  - KEN_EVENT: The handler needs to perform a tunable operation.
  - KEN_BACKOUT: The handler needs to reverse a tunable operation.
  - KEN_UNREGISTER: The handler is being unregistered.

- **instance**
  ID of this particular invocation of this handler.

- **unused**
  The handler data is not used in this handler.

- **event_data**
  When the handler is invoked with `reason` set to KEN_EVENT or KEN_BACKOUT, this parameter points to a `ktune_event_t` structure describing the tunable operation to be performed or reversed. For other reason values, this parameter is NULL.

- **result**
  Points to an integer variable into which the handler must store its result code.

**DESCRIPTION**

This is a generic tunable handler which validates that the tunable is a power of 2. This tunable handler requires the tunable to be of integer type. This tunable handler supports the KTOP_VALIDATE operation. If the result parameter points to the value zero after the handler is called, the tunable is a power of 2. Note that this handler must not be explicitly registered if the powerof2 constraint is specified in the tunable definition in the modmeta file.

**RETURN VALUES**

- **KEN_DONE**
  Handler functions always return KEN_DONE.

**CONSTRAINTS**

See `ktune_handler(9F)` for information on constraints for tunable handlers.

**EXAMPLES**

```c
#include <sys/ktune.h>
{
    int ret;
    ...
    ret = ktune_register_handler(KTUNE_VERSION, "mytunable",
```
0, KEN_UNORDERED,
"power of 2 handler for mytunable",
(ktune_handler_t)ktune_validate_powerof2,
NULL);

...
NAME

ktune_validate_zero_or_min - Handler to allow zero as a valid value for the tunable.

SYNOPSIS

#include <sys/ktune.h>

int ktune_validate_zero_or_min(
    ken_id_t eventid,
    int reason,
    ken_instance_t instance,
    uint64_t *minimum,
    ktune_event_t *event_data,
    int *result
);

PARAMETERS

eventid The ID of the event.
reason Reason the handler was invoked, one of:
KEN_REGISTER The handler is being registered.
KEN_EVENT The handler needs to perform a tunable operation.
KEN_BACKOUT The handler needs to reverse a tunable operation.
KEN_UNREGISTER The handler is being unregistered.
instance ID of this particular invocation of this handler.
minimum The alternate minimum value for the tunable.
event_data When the handler is invoked with reason set to KEN_EVENT or KEN_BACKOUT, this parameter points to a ktune_event_t structure describing the tunable operation to be performed or reversed. For other reason values, this parameter is NULL.
result Points to an integer variable into which the handler must store its result code.

DESCRIPTION

This is a generic tunable handler that allows the tunable to have the value zero even if it falls outside the valid range for the tunable. This handler requires the tunable to be of integer type. This tunable handler supports the KTOP_VALIDATE operation. If the result parameter points to the value zero after the handler is called, tunable value is valid. Note that this handler must not be explicitly registered if the zero_ok constraint is specified in the tunable definition in the modmeta file.

RETURN VALUES

KEN_DONE Handler functions always return KEN_DONE.

CONSTRAINTS

See ktune_handler(9F) for information on constraints for tunable handlers.

EXAMPLES

#include <sys/ktune.h>
{
    int ret;
    uint64_t minimum=10;
ret = ktune_register_handler(KTUNE_VERSION, "mytunable",
    0, KEN_UNORDERED,
    "zero_ok handler for mytunable",
    (ktune_handler_t)ktune_validate_zero_or_min,
    &(minimum));

SEE ALSO

ktune_handler(9F), ktune_event_t(9F), ktune_register_handler(9F).
NAME

ktune_warning - Log a tunable error message.

SYNOPSIS

#include <sys/ktune.h>

void ktune_warning(
    ktune_txn_id_t txnid,  
    const char *format,  
    ...  
);

PARAMETERS

txnid  ID of the transaction in flight, if any.
format  printf-style format string.
...  Arguments needed for format string.

DESCRIPTION

This function logs a tunable change warning message. If a tunable change transaction is in progress, the warning message will be returned to the caller. Otherwise, it will be logged to the system message buffer. This function must only be called from a tunable handler.

Do not embed tunable names in warning message strings. Instead, put a %s where the name is to go, and use ktune_name as an argument to fill in the name.

RETURN VALUES

None

CONSTRAINTS

This function must only be called from a tunable handler.

EXAMPLES

/* From within a tunable handler: */
kttune_warning(event_data->kte_txnid,  
                "%lu is probably not a desirable value for %s.",  
                value, ktune_name(event_data->kte_tuneid));

SEE ALSO

ktune_error(9F), ktune_event_t(9F), ktune_handler(9F), ktune_id(9F), ktune_name(9F).
NAME
ldsid - 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.
Use ldsid with kernel virtual addresses only. Do not call ldsid for a user space address.

RETURN VALUES
The ldsid function returns the returns the space ID (SID) for a kernel virtual address.

CONSTRAINTS
None

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

SEE ALSO
privlbcopy(9F)
major(9F)

NAME
major - Extract the major number from a device number.

SYNOPSIS
#include <sys/sysmacros.h>

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

PARAMETERS
x  A dev_t device number.

DESCRIPTION
The major kernel macro returns the major number from the device number.

CONSTRAINTS
None

SEE ALSO
minor(9F)
makedev(9F)

NAME
makedev - Make a device number from major and minor numbers.

SYNOPSIS
#include <sys/sysmacros.h>

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

PARAMETERS
x  A major number.
y  A minor number.

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

CONSTRAINTS
None

SEE ALSO
major(9F), minor(9F)
MALLOCLNAME

MALLOCCKernel 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 space pointer 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 must 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.

You must set the M_NOWAIT flag if MALLOC 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 must be passed as the flags parameter if the caller is able to block and wait for memory to become available.

RETURN VALUES

The MALLOC returns the following values to the space pointer:

<>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.
• No spinlocks can be held.
EXAMPLE

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.
        */
}

SEE ALSO

FREE(9F), kfree(9F), kmalloc(9F)
map_mem_to_host(9F)

NAME

map_mem_to_host - Map physical bus address to host virtual space.

SYNOPSIS

#include <sys/wsio.h>
caddr_t map_mem_to_host(
    struct isc_table_type *isc,
    caddr_t phys_addr,
    size_t size
);

PARAMETERS

isc 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 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. The map_mem_to_host function returns 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

None

SEE ALSO

unmap_mem_from_host(9F), kernel_iomap(9F), kernel_iounmap(9F)
minor(9F)

NAME

minor - Extract the minor number from a device number.

SYNOPSIS

#include <sys/sysmacros.h>

#define minor(x) ((long)((x)&0xffffffff))

PARAMETERS

x  A dev_t device number.

DESCRIPTION

The minor kernel macro returns the minor number from a device number.

CONSTRAINTS

None

SEE ALSO

major(9F)
NAME

minphys - Limit the \texttt{b\_bcount} field in a \texttt{buf} structure to the value MAXPHYS.

SYNOPSIS

\verb+#include <sys/buf.h>+#

\verb+void minphys(\+
\verb+    struct buf *bp\+
\verb+);+#

PARAMETERS

\texttt{bp}  Pointer to a \texttt{buf} structure.

DESCRIPTION

The \texttt{minphys} kernel function compares \texttt{bp\rightarrow b\_bcount} against the MAXPHYS value defined in \texttt{<sys/param.h>}. If \texttt{bp\rightarrow b\_bcount} is greater than MAXPHYS, \texttt{bp\rightarrow b\_bcount} is changed to MAXPHYS.

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

RETURN VALUES

None

CONSTRAINTS

None

EXAMPLE

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

SEE ALSO

\texttt{physio(9F)}, \texttt{driver\_minphys(9E)}
msg_printf(9F)

NAME

msg_printf - 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.

DESCRIPTION

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

The msg_printf function writes diagnostic information to the msgbuf kernel message buffer only and not to the console.

The msg_printf kernel 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 printf(3S). Other formats specified in printf(3S) are not supported.

RETURN VALUES

The msg_printf function returns the length of the formatted string.

CONSTRAINTS

None

SEE ALSO

printf(9F), printf(3S), vsprintf(9F), vsnprintf(9F), sprintf(3S)
owns_spinlock(9F)

NAME

owns_spinlock - 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

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

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 spin while waiting; they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

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

None

EXAMPLE

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(9F), cspinlock(9F), dealloc_spinlock(9F), get_sleep_lock(9F), spinlock(9F), spinunlock(9F)
NAME

p_pgrp - Return the process group identifier for a process.

SYNOPSIS

#include <sys/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

None

EXAMPLE

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

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

SEE ALSO

gsignal(9F)
### NAME

panic - Soft-crash the operating system.

### SYNOPSIS

```c
#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.

The `panic` function prints the processor status register, the program counter register, the trap type on processor exceptions, and part of the kernel stack.

### RETURN VALUES

The `panic` function does not return.

### CONSTRAINTS

None

### WARNING

The `panic` function halts the system and may cause file system damage. Use this call only to flag catastrophic and unrecoverable failures.
NAME
physio - Perform unbuffered, physical I/O data transfers.

SYNOPSIS
#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. The physio function 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. The strat function reports an error to physio by setting B_ERROR in bp->b_flags and putting an errno value in bp->b_error. See the 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. The physio function 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 minphys kernel supplied function as this parameter.

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

uio iov Pointer to an iovec structure that contains the base address (iov_base) and transfer length (iov_len) of the I/O request.

uio iovcnt Number of iovec structures. If >1, uio iov points to an array of iovec structures.

uio offset Offset into device.
**uio_seg** Type of memory segment to transfer. If set to UIOSEG_USER, physio must be called in the user context.

**uio_resid** Number of bytes of data remaining to be transferred.

For each iovec structure, physio performs the following:

1. Validates the user has appropriate access permissions for the data pages specified by iovec.
2. Sets 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 `uio_offset`.
3. Calls `mincnt` to adjust the transfer length, if too large. If adjusted, physio will make multiple calls to `strat` until all the data specified by `iov_len` has been transformed (or an error occurs).
4. If `uio_seg` is not UIOSEG_KERNEL, locks down the data pages to be transferred.
5. If `uio_seg` is not UIOSEG_KERNEL and the driver has set `C_MAP_BUFFER_TO_KERNEL` in the `d_flags` field of its `drv_ops_t` structure, physio maps the user buffer into kernel space. This additional overhead is performed for legacy drivers that do not know how to access user space in their strategy function.
6. Calls strat passing the buf structure pointer `bp` as a parameter, then waits for the I/O request to complete by calling `biowait`.
7. The driver calls `biodone` when the I/O request completes to awaken the thread waiting in `biowait`.
8. After the I/O request has completed, data pages that have been locked down are unlocked.
9. The following fields in the uio structure are updated:
   - `uio_resid` is decreased by the transfer length.
   - `uio_offset` is increased by the transfer length.

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

**RETURN VALUES**

0 Successful completion.

<>0 Error.

**CONSTRAINTS**

Do not call in an interrupt context.

Do not call while holding a spinlock.

**EXAMPLE**

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

**SEE ALSO**

`biodone(9F), biowait(9F), buf(9F), minphys(9F), uio(9F)`
NAME

printf - 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)).

The printf kernel function writes diagnostic information to the console and into the msgbuf kernel message buffer.

The printf kernel 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 a 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 printf(3S). Other formats specified in printf(3S) are not supported.

The printf kernel function is buffered. Therefore, console error messages may not be synchronous with the current kernel state.

RETURN VALUES

The printf kernel function returns the length of the formatted string.

CONSTRAINTS

None

SEE ALSO

vsprintf(9F), vsnprintf(9F), msg_printf(9F), printf(3S)
privileged_cred(9F)

NAME
privileged_cred - Determine if a credential has a specified privilege asserted.

SYNOPSIS
#include <sys/privileges.h>
#include <sys/priv.h>

int privileged_cred(
    const cred_t *cr,
    priv_t priv
);

PARAMETERS

- **cr**: Credential to be checked.
- **priv**: Privilege for which to check whether it is asserted.

DESCRIPTION
The `privileged_cred` function determines whether the credential has the privilege `priv` asserted. A privilege can be asserted in the following ways:

- The privilege is in the effective privilege set of the thread.
- The thread has an effective `uid` equal to 0, `priv` is part of a root replacement privilege, and if compartments are enabled, `priv` is not disabled in the thread’s compartment.
- The privilege is granted to all processes (not implemented).

RETURN VALUES
The `privileged_cred` function returns the following values:

- **0**: Failure. The privilege is not enabled or the credential is null.
- **1**: Success. The privilege is enabled.

NOTES
The `privileged_cred` function does not update the ASU bit and does not generate audit information.

LOCKING CONSIDERATIONS
The `privileged_cred` function can call `printf` to generate diagnostic messages. The `printf` function uses a spinlock of order `PRINTF_LOCK_ORDER`. This means that the callers of these functions must not hold a spinlock of order `PRINTF_LOCK_ORDER` or higher.

EXAMPLES
The following example uses `drv_priv` to unlock a BTLAN driver request:

```c
if (drv_priv(ioc_reqp->ioc_cr) != 0) {
    /* not superuser */
    status = EPERM;
    break;
}
```

The following example uses `privileged_cred` to unlock a BTLAN driver request:

```c
if (!privileged_cred(ioc_reqp->ioc_cr, PRIV_NETADMIN)) {
    /* unprivileged user */
    status = EPERM;
    break;
}
```
SEE ALSO

NAME

privlbcopy - 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, use ldsid to get the SID of the address. Do not call privlbcopy to copy data from or to user space without first calling physio. The physio function validates access to the user buffer, locks the corresponding data pages in memory, puts the user SID and address in the b_spaddr and b_un.b_addr fields of the buf structure. (b_spaddr and b_un.b_addr will contain a kernel space ID and address if the driver specifies C_MAP_BUFFER_TO_KERNEL in its drv_ops_t structure.)

RETURN VALUES

0 Successful completion.
<>0 Error.

CONSTRAINTS

None

EXAMPLE

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

**WARNING**

Do not use privlbcopy 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, a High Priority Machine Check (HPMC), or a call to panic.

**SEE ALSO**

bcopy(9F), copyin(9F), copyout(9F), ldsid(9F), physio(9F)
NAME

psignal - Send the specified signal to a process.

SYNOPSIS

#include <sys/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>.

The psignal function 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

None

EXAMPLE

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

/*
 * In the user context, save the pointer to the proc structure.
 */
proc_t *my_procp = u.u_procp;
...

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

SEE ALSO

gsignal(9F)
NAME
putc - 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
None

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

/* concatenate characters from one clist to another */
/* expects putc to succeed */
{
    int c;

    while ((c = getc(from)) >= 0)
        if(putc(c, to))

SEE ALSO
getc(9F), getcb(9F), getcf(9F), putcb(9F), putcf(9F)
NAME

putcb - 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

The putcb kernel function adds the cblock pointed to by cb onto the clist pointed to by list.

RETURN VALUES

None

CONSTRAINTS

None

SEE ALSO

getcb(9F), getc(9F), getcf(9F), putc(9F), putcf(9F)
NAME
putcf - 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
The putcf kernel function returns the cblock pointed to by cb to the free list.

RETURN VALUES
None

CONSTRAINTS
None

EXAMPLE

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

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

    ...
}

SEE ALSO
getcf(9F), getc(9F), getcb(9F), putc(9F), putcb(9F)
NAME

seltrue - Select driver entry point that returns true for traditional select flags.

SYNOPSIS

#include <sys/select.h>

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

The seltrue function 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

None

EXAMPLES

SEE ALSO

select(2)
NAME

selwakeup - Wake up a kernel thread sleeping on a select condition.

SYNOPSIS

#include <sys/select.h>

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

Do not call while holding a spinlock of order >= SCHED_LOCK_ORDER.

EXAMPLE

#include <sys/select.h>
#define MYSEL_COLL 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

waiting_in_select(9F)
sleep(9F)

NAME

sleep - Sleep on a channel.

SYNOPSIS

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

int sleep(
    void *chan,
    int  pri
);
```

PARAMETERS

chn Channel (or kernel address) on which to sleep.

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 wake up events and `chan`. The `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` might 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` might 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 `sleep`, 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

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

Do not call in an interrupt context.
Do not call while holding a spinlock.

EXAMPLE

/*
 * 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(9F), wakeup(9F),

180 Kernel Reference Pages
NAME

spinlock - Acquire (lock) a spinlock.

SYNOPSIS

```c
#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 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 (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

None

EXAMPLE

```c
/*
 * 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(9F), cspinlock(9F), dealloc_spinlock(9F), get_sleep_lock(9F), owns_spinlock(9F),
spinunlock(9F)
NAME
spinunlock - Release (unlock) a spinlock.

SYNOPSIS
#include <sys/spinlock.h>
void spinunlock(
    lock_t *lock
);

PARAMETERS
lock Pointer to a lock_t structure.

DESCRIPTION
The spinunlock kernel function releases (unlocks) the spinlock pointed to by lock.
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 spin while waiting; they
do not block. For the duration that a processor owns a spinlock, external interrupts to the processor
are disabled.

RETURN VALUES
None

CONSTRAINTS
None

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(9F), cspinlock(9F), dealloc_spinlock(9F), owns_spinlock(9F), spinlock(9F)
NAME

sprintf - Kernel version of sprintf.

SYNOPSIS

#include <sys/kern_svcs.h>

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

PARAMETERS

str Address of buffer to hold the formatted string.
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 identical to the C library sprintf routine (see sprintf(3S)).
The 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>
<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 sprintf(3S). Other formats specified in sprintf(3S) are not supported.

RETURN VALUES

The sprintf function returns the length of formatted string.

CONSTRAINTS

None

SEE ALSO

msg_printf(9F),printf(9F),sprintf(3S)
NAME

strcat - Concatenate two strings.

SYNOPSIS

#include <sys/kern_svc.h>

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

PARAMETERS

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

DESCRIPTION

The strcat kernel function appends string buf to string str.

RETURN VALUES

The strcat function returns buf.

CONSTRAINTS

None

SEE ALSO

strcpy(9F), strlen(9F), strncmp(9F), strncpy(9F)
**NAME**

strcmp - Compare two strings.

**SYNOPSIS**

```c
#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**

None

**EXAMPLE**

```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";
    }
    ...
    ...
}
```

186 Kernel Reference Pages
SEE ALSO

strcpy(9F), strlen(9F), strncmp(9F), strncpy(9F)
**NAME**

strcpy - Copy the characters from one string to another string.

**SYNOPSIS**

```c
#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**

The `strcpy` function returns the string pointed to by `s1`.

**CONSTRAINTS**

None

**EXAMPLE**

```c
{
    .
    .
    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(9F), strlen(9F), strncmp(9F), strncpy(9F)`
NAME
strlen - Gets the number of nonnull bytes in a string.

SYNOPSIS
#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
The strlen function returns the number of non-null bytes in the string.

CONSTRAINTS
None

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(9F), strcpy(9F), strncmp(9F), strncpy(9F)
NAME

strncmp - Compare the first n characters of two strings.

SYNOPSIS

#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

None

EXAMPLE

{

  .
  .
  if (strncmp(driver_name, "enet",4) !=0)
      /* The driver is not ENET */
      return -1;
  else {
      .
      .
  }
  .

SEE ALSO

strcmp(9F), strlen(9F), strncmp(9F), strncpy(9F)
strncpy(9F)

NAME

strncpy - 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. The strncpy function returns a pointer to the copied (s1) string.

CONSTRAINTS

None

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(9F), strlen(9F), strncmp(9F), strcpy(9F)
NAME
suser - 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
sw_trigger(9F)

NAME

sw_trigger - 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 cannot 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
The sw_trigger function is a void function.

CONSTRAINTS
None

EXAMPLE
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
```
timeout(9F)

NAME

timeout - 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 two 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 integer 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 to expire. The timeout can be canceled by making a corresponding call to untimeout.

RETURN VALUES

The timeout function returns a pointer to a callout structure. The caller must treat it as an opaque handle, and not try to change the contents of the structure.

CONSTRAINTS

Do not call while holding a spinlock of order >= CALLOUT_LOCK_ORDER.

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)
{
    ...
}
WARNINGS

Each call to timeout allocates a callout resource, and the resource is not released until the time value expires or the timeout is canceled. The kernel may panic if it has no more callout resources available, and is unable to dynamically allocate more.

SEE ALSO

un timeout(9F), delay(9F).
NAME

uio - 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). The `uio` structure 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 must not access fields in the `uio` structure directly. The `physio` kernel function 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>`, and has the following fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uio_iov</code></td>
<td>struct <code>iovec</code> *</td>
</tr>
<tr>
<td><code>uio_iovcnt</code></td>
<td>size_t</td>
</tr>
<tr>
<td><code>uio_seg</code></td>
<td>uint32_t</td>
</tr>
<tr>
<td><code>uio_resid</code></td>
<td>long</td>
</tr>
<tr>
<td><code>uio_fpflags</code></td>
<td>uint32_t</td>
</tr>
<tr>
<td><code>uio_offset</code></td>
<td>off_t</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 must not wait:

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

- `FNBLOCK` The driver must not wait if the requested data transfer can not be done immediately. The request must be terminated and return `EAGAIN` as the error code.
**uiomove**

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

**SEE ALSO**

iovec(9S), open(2), physio(9F), read(2), readv(2), uiomove(9F), write(2), writev(2)
uiomove(9F)

NAME

uiomove - 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 or 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. The uiomove function 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. The uiomove function 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.

EXAMPLE

```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(9F), geteblk(9F)`
NAME

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

RETURN VALUES

The map_mem_to_host function returns an indeterminate value. Treat it as if it was a void function.

CONSTRAINTS

None

SEE ALSO

map_mem_to_host(9F)
NAME

untimeout - 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. 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 canceled by taking it off the list and releasing allocated resources.

RETURN VALUES

>=0   Number of clock ticks remaining in the cancelled timeout request.
-1    Timeout request not found. This is a normal return if the timeout request has already expired or has been dispatched to be executed.

CONSTRAINTS

Do not call while holding a spinlock of order >= CALLOUT_LOCK_ORDER.

EXAMPLES

/*
 * Cancel a previous timeout to call my_timeout_func()
 */
(void) untimeout(my_timeout_func, my_arg);

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

SEE ALSO

timeout(9F).
user_iomap(9F)

NAME

user_iomap - Maps I/O memory to a user address.

SYNOPSIS

#include <sys/iomap.h>

void *user_iomap(
    void *vaddr,
    void *paddr,
    pgcnt_t count
);

PARAMETERS

vaddr       Virtual address requested. must be NULL.
paddr       Physical address of the I/O memory location from the view of the processor.
count       Count of base pages to map.

DESCRIPTION

The user_iomap function memory maps into shared user address space a range of contiguous pages in I/O address space starting at the physical address paddr for count pages. The value returned is the user virtual address that maps onto the base of the I/O memory location specified by paddr and count.

The virtual address returned will be appropriate to the user process: a 32-bit address is returned for 32-bit processes and a 64-bit address is returned for 64-bit processes. The call to user_iomap must be made in the context of the user process; the call cannot be made in interrupt context.

The physical address is the 64-bit physical address from the view of the processor, and the address must be pagesize aligned. The pagesize can be obtained by calling iomap_pagesize.

The 64-bit physical address paddr is not necessarily the same address value that is obtained from the Base Address Register (BAR) of a PCI device. The caller may need to convert the BAR value to the appropriate 64-bit physical address value used by the processor.

You can call the user_iomap function multiple times for the same I/O memory range to allow access to the range by multiple user processes. For each call made to user_iomap, there must be a corresponding call made to user_iounmap in the context of each user process before the calling module unloads, or the module must wait for each user process to terminate, in which case the unmapping is done by the kernel.

It is normal for a call to user_iomap to fail. Possible reasons for failure include:

• The pages are mapped for kernel-only access with PROT_KRW and the user virtual address is equivalent to the kernel virtual address. If the pages are mapped by the WSIO driver environment or mapped using wsio_map_reg, protection is set to PROT_KRW.

• The pages overlap with an I/O memory range that has already been mapped by a previous call to user_iomap or kernel_iomap. To successfully map an I/O memory range multiple times, the mappings must cover exactly the same range (same paddr and count parameters).

A call to user_iomap might also fail if the driver previously called wsio_map_reg or if the WSIO driver environment has mapped the I/O memory on behalf of the driver. If the driver's isc->if_reg_ptr is not NULL, WSIO has mapped for kernel-only access (up to 8 KB) of the first PCI BAR in the device that specifies a memory range.

RETURN VALUES

<>NULL    User virtual address.
NULL Error. Mapping failed.

CONSTRAINTS
Must be called in the user context.
Do not call in interrupt context.
Do not call while holding a spinlock.

EXAMPLES

```c
void  *physaddr;
void  *useraddr;
size_t pagesize;

/*
 * Get the physical address as seen by the processor.
 * Verify that the physical address is pagesize aligned.
 */
physaddr = xyz;
pagesize = iomap_pagesize();
VASSERT(((uintptr_t)physaddr % pagesize) == 0);

/*
 * Memory map the user virtual address onto a page of the
 * I/O memory location.
 */
useraddr = user_iomap(NULL, physaddr, 1L);

if (useraddr == NULL) {
    /*
     * Failed to map user virtual address.
     * Add error handling code here.
     */
}
```

SEE ALSO
iomap_pagesize(9F), kernel_iomap(9F), kernel_iounmap(9F), user_iounmap(9F).
user_iounmap(9F)

NAME

user_iounmap - Unmap I/O memory for the assigned user address.

SYNOPSIS

#include <sys/iomap.h>

int user_iounmap(
    void *vaddr,
    pgcnt_t count
);

PARAMETERS

vaddr  Virtual address returned by user_iomap.

count  Count of pages to unmap.

DESCRIPTION

The user_iounmap function removes the user virtual address assigned to map a range of I/O memory pages. If the same virtual address is assigned to multiple user processes, the address will remain valid for the other user processes.

The vaddr parameter is the virtual address returned by user_iomap, and the count parameters must match the count parameter passed to user_iomap.

RETURN VALUES

0  Success.

1  Failure.

CONSTRAINTS

Must be called in the user context.

Do not call in interrupt context.

Do not call while holding a spinlock.

EXAMPLES

void   *physaddr;
void   *useraddr;
pgcnt_t count;
int     retval;

...

/*
 * Memory map the user virtual address onto the I/O memory
 * location for count number of pages.
 */
useraddr = user_iomap(NULL, physaddr, count);

...

/*
 * Remove the mapping for the user virtual address.
 */
retval = user_iounmap(useraddr, count);
VASSERT(retval == 0);
SEE ALSO

iomap_pagesize(9F), kernel_iomap(9F), kernel_iounmap(9F), user_iomap(9F).
NAME
VASSERT - Test an assertion on kernels compiled with the OSDEBUG flag.

SYNOPSIS
#include <sys/debug.h>

VASSERT (expr);

PARAMETERS
expr An expression that evaluates to true or false.

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

RETURN VALUES
None

CONSTRAINTS
None

SEE ALSO
panic(9F)
NAME

waiting_in_select - Determines if a kernel thread is waiting on a select condition.

SYNOPSIS

#include <sys/select.h>

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

The waiting_in_select function returns the following values:

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

CONSTRAINTS

None

EXAMPLES

#include <sys/select.h>

#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(9F)
**NAME**

wakeup - Wake up all threads sleeping on a channel.

**SYNOPSIS**

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

int wakeup(
    void *chan
);
```

**PARAMETERS**

*chan* Channel passed in the corresponding call to `sleep`.

**DESCRIPTION**

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

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

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

**RETURN VALUES**

The `wakeup` returns the number of threads awakened.

**CONSTRAINTS**

Do not call while holding a spinlock of order greater than `SLEEP_Q_LOCK_ORDER`.

**EXAMPLES**

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

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

**SEE ALSO**

`get_sleep_lock(9F)`, `sleep(9F)`
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. See Appendix A (page 629) for a list.
NAME

bp_dma_cleanup - 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

None

EXAMPLE

```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(9F), dma_cleanup(9F), dma_setup(9F)`
NAME

bp_dma_setup - 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 function sets up a DMA transfer for a list of buffers. A driver calls this function when bp->b_merge_cnt is not zero (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

None

EXAMPLES

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_count;
    retval = dma_setup(isc, dma_parms);
} else {
    retval = bp_dma_setup(isc, bp, bp_dma_parms);
}

SEE ALSO

dma_cleanup(9F), dma_setup(9F), bp_dma_cleanup(9F)
NAME
dma_cleanup – Clean up from a DMA transfer.

SYNOPSIS
#include <sys/wsio.h>
int dma_cleanup(
    struct isc_table_type *isc,
    struct dma_parms *dma_parms
);

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

DESCRIPTION
The dma_cleanup WSIO function performs the required cleanup for a DMA transfer.

RETURN VALUES
None

CONSTRAINTS
None

SEE ALSO
bp_dma_setup(9F), dma_setup(9F)
NAME

dma_parms - 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;
}

#include <sys/eisa.h>

/* dma_options bits */
#define DMA_ISA 0x1
#define DMA_TYPEA 0x2
#define DMA_TYPEB 0x4
#define DMA_BURST 0x8
#define DMA_TYPEC DMA_BURST
#define DMA_DEMAND 0x10
#define DMA_SINGLE 0x20
#define DMA_BLOCK 0x40
#define DMA_CASCADE 0x80
#define DMA_8BYTE 0x100
#define DMA_16WORD 0x200
#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
dma_setup and dma_cleanup kernel routines use a pointer to this structure as one of the
parameters.

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

SEE ALSO

bp_dma_cleanup(9F), bp_dma_setup(9F), dma_cleanup(9F), dma_setup(9F), iovec(9S), wsio_map(9F)
NAME
dma_setup - Set up a DMA transfer.

SYNOPSIS
#include <sys/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
None

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(9F), dma_cleanup(9F)
NAME
dma_sync_IO - 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.
   IO_SYNC_FORCPU  Same as IO_READ.
   IO_SYNC_FORDEV  Same as IO_WRITE.
   IO_SYNC_MEM     Synchronize processor caches with host memory; caches are flushed to memory when used with IO_WRITE (even on coherent platforms).
   IO_WRITE        Flush processor caches for outbound data on noncoherent platforms.

DESCRIPTION
Drivers call dma_sync_IO to synchronize the processor caches with DMA transactions mastered by their devices. The dma_sync_IO function 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:

1. 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_SYN` hint must also be set to reduce the performance penalty of synchronizing the cache flushes on noncoherent platforms.

2. Prior to start a read transaction:
   For each buffer that is to be read into, the driver must call `dma_sync_IO` with the `IO_READ_START` hint set. On noncoherent platforms, this will cause the associated processor caches to be purged. For all but the last buffers, the `IO_NO_SNYC` hint must also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms.

3. 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 must 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**

None

**SEE ALSO**
driver_addr_probe(9F)

NAME

driver_addr_probe - Interface driver address probing function.

SYNOPSIS

#include <sys/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 structure.
dev_probe Probe function registered by device driver to be called by driver_addr_probe.
drv_info The drv_info_t structure 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. Register this function with WSIO by executing the wsio_register_addr_probe routine as part of the interface driver_attach routine. Commonly, driver is replaced by your driver’s name.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

PROBE_SUCCESS Successfully found something it identified.
PROBE_UNSUCCESSFUL Got to end of the appropriate address range without finding anything, or something went wrong with the probe.
CONSTRAINTS

None

SEE ALSO

driver_class_probe(9E), wsio_probe_dev_info(9F), wsio_register_probe_func(9F),
wsio_register_addr_probe(9F)
driver_attach(9E)

NAME
driver_attach - Claim a device for a driver.

SYNOPSIS

int driver_attach(
    uint32_t card_id,
    struct ics_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. 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 the 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 (for example, 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, test this flag to see if there was a prior claim.

See the HP-UX 11i v2 Driver Development Guide for information.

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

None

EXAMPLE

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, use code similar to the following example in determining that the device is the drivers':

```c
int zzz_attach(uint32_t idparm, struct ics_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(9F), isc_table_type(9F)
driver_close(9E)

NAME

driver_close - 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(9F) and minor(9F)).
(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(9E).

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

DESCRIPTION

The driver_close WSIO function is provided by the driver writer. It can have any unique
name. Pass the name to WSIO Services by specifying it in the d_close field of the drv_ops
structure. Commonly, driver is replaced by the driver's name.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

The file system function which calls the 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 the 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 must return some
integer value.

CONSTRAINTS

None

SEE ALSO

close(2), driver_open(9E), drv_ops(9S), open(2)
NAME

driver_dev_init – 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. 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 the driver’s name.

See the HP-UX 11i v2 Driver Development Guide for information.

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

None

SEE ALSO

driver_install(9E)
driver_dev_probe(9E)

NAME

driver_dev_probe - Interface driver device probing function.

SYNOPSIS

#include <sys/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 structure.
drv_info The drv_into_t structure 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 one is needed, register it with WSIO by executing the wsio_register_dev_probe routine as part of the driver_install routine. Commonly, driver is replaced by your driver's name.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

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

CONSTRAINTS

None
SEE ALSO

driver_addr_probe(9E), wsio_register_addr_probe(9F), wsio_probe_dev_info(9F),
wsio_register_dev_func(9F)
**NAME**
driver_if_init -- 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. Pass the name to WSIO Services by specifying it in the isc->gfsw->init of the ISC structure with driver's driver_attach routine. Commonly, driver is replaced by the driver's name.

See the HP-UX 11i v2 Driver Development Guide for information.

**RETURN VALUES**
0 Successful completion.
-1 Error

**CONSTRAINTS**
None

**SEE ALSO**
driver_attach(9E), isc_table_type(9F),
NAME

driver_install - 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 the driver.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

None. If the routine encounters an error, it must print an error message and return.

CONSTRAINTS

None

SEE ALSO

config(1M), driver_attach(9E), driver_dev_init(9E), driver_install(9E), driver_probe(9E), install_driver(9S), master(4), wsio_install_driver(9F), wsio_register_addr_probe(9F), wsio_registerProbe_func(9F)
driver_ioctl(9E)

NAME
driver_ioctl - 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. Pass the name to WSIO Services by specifying it in the d_ioctl field of the drv_ops structure. Commonly, driver is replaced by the driver's name.
See the HP-UX 11i v2 Driver Development Guide for information.

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

CONSTRAINTS
None

LP64 CONSIDERATIONS
Pay particular attention to the cmd parameter, which has different values depending upon the calling program environment. See the HP-UX 11i v2 Driver Development Guide for more information.

SEE ALSO
drv_ops(9S), errno(2), ioctl(2), ioctl(5)
**NAME**

driver_isr - Execute device interrupt in interrupt context.

**SYNOPSIS**

```c
int driver_isr(
    uintptr_t arg
);
```

**PARAMETERS**

arg Parameter passed in with the wsio_intr_alloc call.

**DESCRIPTION**

The driver_isr WSIO function is provided by the driver writer, it can have any unique name. Pass the name to WSIO Services by specifying it as a parameter of the wsio_intr_alloc function, executed in the driver_attach or driver_if_init routine. Commonly, driver is replaced by the driver's name.

See the HP-UX 11i v2 Driver Development Guide for information.

**RETURN VALUES**

0 The card does not belong to this driver.
1 This routine handled the interrupt.

**CONSTRAINTS**

None

**SEE ALSO**

wsio_intr_alloc(9F)
NAME
driver_load - Called by the DLKM infrastructure while loading the driver dynamically.

SYNOPSIS
int driver_load(
    void *drv_infopg
);

PARAMETERS
drv_infop A pointer to the drv_info structure for the driver.

DESCRIPTION
A pointer to the driver_load function is passed to the DLKM infrastructure through
modwrapper structure. The name of the driver_load function must be driver_load.
The driver_load function must perform all the functions that are performed by the
driver_install function except for the following differences:

• The driver_load function for a PCI interface driver must call
  mod_wsio_attach_list_add to add its attach function, if any, to the “DLKM attach list”
  instead of inserting its driver_attach at the head of the global pci_attach chain.
• The driver_load function must perform any driver specific initialization and must not
  add its driver_init function to the dev_init chain.
• Call wsio_activate_probe to activate any probe routine registered in the load routine.

NOTE:  Add the static keyword at the end of initfunc driver_install directive in the
modmeta file to prevent the kernel configuration from calling both driver_install and
driver_load() routines.

RETURN VALUES
On success, driver_load returns 0, otherwise it returns an error number from <sys/errno.h>
file to indicate an error.

EXAMPLES
The following is an example of a WSIO interface driver load function:
#include <sys/wsio.h>
#include <sys/errno.h>
#include <sys/moddefs.h>

static wsio_drv_info_t driver_wsio_info = { ... };
static int driver_pci_attach (...);
static void driver_init(...);
int driver_load(void *arg)
{
    /* Pass the drv_info passed to this function to wsio_install_driver */
    driver_wsio_info.drv_info = (drv_info_t *) arg;

    /* Register the driver with WSIO */
    if (!wsio_install_driver(&driver_wsio_info)) {
        printf("<driver>: wsio_install_driver failed!!\n");
        goto err3;
    }
    /* Perform driver-specific initialization, but do not call the next
    * function in the dev_init list */
(void) driver_init();
/* Add the attach function to the DLKM attach list */
if (mod_wsio_attach_list_add(MOD_WSIO_PCI, &driver_pci_attach) != 0) {
    printf("<driver>: mod_wsio_attach_list_add failed!!\n");
    goto err2;
}
/* Register the device probe */
if (wsio_register_dev_probe(IF_CLASS, driver_probe, "probe_name")
    != 0) {
    printf("<driver>: wsio_register_dev_probe failed!!\n");
    goto err1;
}
/* Attach the probe function to the drv_info structure */
if (wsio_activate_probe("probe_name", driver_wsio_info.drv_info)
    != 0) {
    printf("<driver>: wsio_activate_probe failed!!\n");
    (void) wsio_unregister_probe(IF_CLASS, "probe_name");
    goto err1;
}
return (0);
err1:
    (void) mod_wsio_attach_list_remove(MOD_WSIO_PCI, &driver_pci_attach);
err2:
    (void) wsio_uninstall_driver(&driver_wsio_info);
err3:
return (ENXIO);
}

SEE ALSO

driver_install(9E), driver_unload(9E), modwrapper(9F), kcmodule(1M), wsio_install_driver(9F),
wsio_uninstall_driver(9F), driver_init(9E), mod_wsio_attach_list_add(9F),
mod_wsio_attach_list_remove(9F), wsio_register_dev_probe(9F), wsio_activate_probe(9F),
wsio_unregister_probe(9F), modmeta(4)
NAME

driver_unload - Called by the DLKM Infrastructure during unload of a dynamically unloadable driver

SYNOPSIS

int driver_unload(
    void *drv_infop
);

PARAMETERS

drv_infop A pointer to the drv_info structure for the driver.

DESCRIPTION

A pointer to the driver_unload function is passed to the DLKM infrastructure through modwrapper structure. The name of the driver_unload function must be driver_unload. A driver that declares support for the unload capability in its metadata must provide a driver_unload function.

For the driver to be unloaded, the number of opens on the device file associated with the driver must be zero. The driver_unload function does not need to check for this. The driver's unload routine is not called if there are any outstanding opens on the device special file.

The driver_unload function must call mod_wsio_attach_list_remove to remove its attach function, if its attach function was added to the DLKM attach list in the driver_load function. It must call the wsio_unregister_dev_probe to unregister the driver's probe function, if the probe function was registered with WSIO during the driver_load. The driver_unload function must perform any other necessary driver-specific cleanup operations (for example, clear any outstanding timeouts, free any interrupt objects, unregister ISR routine, free allocated memory resources, free spinlocks, and call ktune_unregister_handler to unregister any tunable handlers that were registered by the tunable_init function). It must call wsio_uninstall_driver passing to it a pointer to the driver's wsio_drv_info structure.

If in any step, the driver_unload fails, the driver must undo any action prior to the failure and return an appropriate error from <sys/errno.h>.

RETURN VALUES

On success, driver_unload returns 0; otherwise, it returns an error number from <sys/errno.h> file to indicate an error.

EXAMPLES

The following is an example of a WSIO interface driver unload function:

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

static wsio_drv_info_t driver_wsio_info = { ... };
static int driver_pci_attach (...);

int driver_unload(void *arg)
{
    driver_wsio_info.drv_info = (drv_info_t *) arg;

    /* Remove the attach function from the DLKM attach list */
    if (mod_wsio_attach_list_remove(MOD_WSIO_PCI, &driver_pci_attach) != 0) {
        printf("<driver>: mod_wsio_attach_list_remove failed!!\n");
        return (ENOMEM);
    }

    /* Unregister the device probe */
    if (wsio_unregister_dev_probe(IF_CLASS, driver_probe, "probe_name") != 0) {
        printf("<driver>: wsio_unregister_dev_probe failed!!\n");
    }

    /* Other cleanup operations */
}
```
mod_wsio_attach_list_add(MOD_WSIO_PCI, driver_pci_attach);
return (ENXIO);
}

/* Perform driver cleanup for each driver instance, find the isc claimed
 * by the driver, clear any pending timeouts, unregister ISR routine, free
 * allocated memory and resources, call ktune_unregister_handler to
 * unregister any handlers that were registered by tunable_init function
 */
if (wsio_uninstall_driver(&driver_wsio_info) != 0) {
  printf("<driver>: wsio_uninstall_driver failed!!\n");
  mod_wsio_attach_list_add(MOD_WSIO_PCI, driver_pci_attach);
  wsio_register_dev_probe(IF_CLASS, driver_probe, "probe_name");
  return (ENXIO);
}
return 0;

SEE ALSO

driver_install(9E), driver_load(1M), modwrapper(1M), kmodule(1M), wsio_install_driver(9F),
wsio_uninstall_driver(9F), driver_init(9E), mod_wsio_attach_list_add(9F),
mod_wsio_attach_list_remove(9F), wsio_register_dev_probe(9F), wsio_activate_probe(9F),
wsio_unregister_probe(9F)
**NAME**

driver_minor_build - Build a minor number.

**SYNOPSIS**

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

int driver_minor_build(
    isc_table_type *isc,
    hw_path_t *dev_path,
    char *option
);
```

**PARAMETERS**

- **isc**: A pointer to the ISC structure associated with the interface card for the device.
- **dev_path**: A pointer to a structure containing device hardware path information relative to the interface card.
- **option**: A null-terminated string indicating device-specific options.

**DESCRIPTION**

The `driver_minor_build` WSIO function is provided by the driver writer, it can have any unique name. 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 the driver's name.

See the *HP-UX 11i v2 Driver Development Guide* for information.

**RETURN VALUES**

- **>0**: Successful completion. The value is the minor number.
- **-1**: Error. The minor number could not be built.

**CONSTRAINTS**

None

**SEE ALSO**

`wsio_drv_data_t(9F)`
driver_minphys(9E)

NAME
driver_minphys — Driver specific transfer size adjustment.

SYNOPSIS
#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. 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 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 driver_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
None

SEE ALSO
minphys(9F), physio(9F)
driver_open(9E)

NAME

driver_open - 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(9F) and
minor(9F).
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 the driver.
The driver, therefore, can usually ignore these flags.
Nevertheless, the kernel translates the O_XXXX values into corresponding FXXXX
values, which it passes to the driver_open routine. The flags of possible interest to
the driver include: FREAD, FWRITE, FNDELAY, and FEXCL.

mode Whether this is a call to a block or char driver. This parameter is not accessible from
an open(2) call.
dummy A parameter used as by some drivers, though it is not accessible from an open(2) call.

DESCRIPTION

The driver_open WSIO function is provided by the driver writer; it can have any unique name.
Pass the name to WSIO Services by specifying it in the driver_open field of the drv_ops
structure. Commonly, driver is replaced by the driver’s name.
See the HP-UX 11i v2 Driver Development Guide for information.

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 must 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**

None

**SEE ALSO**

`drv_ops(9S), open(2)`
NAME

driver_psize - 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. 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 function must 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 the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

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

-1    Error

CONSTRAINTS

None

SEE ALSO

drv_ops(9S)
driver_read(9E)

NAME

driver_read – Read data from and 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. The driver_open routine must 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 iovector 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 the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

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

CONSTRAINTS

None

EXAMPLES

See physio(9F) and uiomove(9F).

SEE ALSO

drv_ops(9S), physio(9F), uiomove(9F)
**driver_select***(9E)***

**NAME**

driver_select - 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  
  - FWRITE  
  - 0  

**DESCRIPTION**

The `driver_select` WSIO function is provided by the driver writer; it can have any unique name. 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 the [HP-UX 11i v2 Driver Development Guide](#) for information.

**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 `driver_select` returns to the user.

- `0`  
  False — The device or driver is not ready for read or write or no exception condition was found. The `driver_select` function 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 must return false and set an error in `u.u_error`. If it detects an error while selecting for an exception condition, it must return true and set an error in `u.u_error`.

**CONSTRAINTS**

None

**SEE ALSO**

- `drv_ops(9S)`  
- `selwakeup(9F)`  
- `select(2)`
driver_strategy(9E)

NAME

driver_strategy - Execute block read or write for character or block devices.

SYNOPSIS

#include <sys/conf.h>

int 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, pass the name to WSIO Services by specifying it in the driver_strategy field of the drv_ops structure. For a character device, pass the name as a parameter to physio. Commonly, driver is replaced by your driver's name.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

None

CONSTRAINTS

None

SEE ALSO

physio(9F)
driver_write(9E)

NAME
driver_write - 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. The driver_open routine must 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 the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES
0 Successful completion.
<>0 Error. The value is expected to be an errno value.

CONSTRAINTS
None

SEE ALSO
drv_ops(9S), physio(9F), uiomove(9F)
NAME
drv_info - Driver information structure.

SYNOPSIS
#include <sys/conf.h>

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

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

The relevant fields are described. All other fields in a drv_info_t must 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 used in the system file (usually /stand/system), in the $DRIVER_INSTALL section of a master file in /usr/conf/master.d, and a 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 (for example, disk, type, and pseudo). 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>
</tbody>
</table>
DRV_SAVE_CONF  Save configuration information to /etc/ioconfig. This file retains potentially volatile information, such as dynamic major numbers and card instance numbers, across reboots.

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

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

**NOTE:**  The values specified for *b_major* and *c_major* override the values entered in a master file in /usr/conf/master.d (see *master*(4)).

**SEE ALSO**

*config*(1M), *driver_install*(9E), *drv_ops*(9S), *wsio_drv_info*(9F)
drv_ops(9S)

NAME

drv_ops - 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 must be NULL. Except as noted, entry points that do not apply to the driver or that the driver does not provide must 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 as shown in Table 2-1 "Device Driver Fields" for drv_ops_t structure types.

Table 2-1 Device Driver Fields

<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 the driver_open routine, which enables a device for subsequent operations.</td>
</tr>
<tr>
<td>d_close</td>
<td>both</td>
<td>Pointer to the 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 the 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 the driver_psize routine. For a swapping device, it must return the size of the swap partition.</td>
</tr>
</tbody>
</table>
### Table 2-1 Device Driver Fields (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_read</td>
<td>character</td>
<td>Pointer to the driver_read routine, which must return the requested data transferred from the device.</td>
</tr>
<tr>
<td>d_write</td>
<td>character</td>
<td>Pointer to the driver_write routine, which must write the requested data to the device.</td>
</tr>
<tr>
<td>d_ioctl</td>
<td>character</td>
<td>Pointer to the driver_ioctl routine, which sends control information to, or gets it from a device.</td>
</tr>
<tr>
<td>d_select</td>
<td>character</td>
<td>Pointer to the driver_select routine, which can be used to test for I/O completion and driver-dependent exception conditions. If the device is always ready for reading or writing, put seltrue in the d_select field. If this happens, calls to select always return true without invoking the driver.</td>
</tr>
<tr>
<td>pfilter</td>
<td>both</td>
<td>Pointer to a pfilter_t structure, use the &amp;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;cpd_pfilter pointer is required for such disks; it is ignored under other conditions (or use NULL).</td>
</tr>
<tr>
<td>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>

The flag bit defines for `d_flags` are:

- **C_ALLCLOSES**: Force a call to driver_close on every closing of the device. (The default action is to call the driver’s close routine only on the last close of the device.)
- **C_NODELAY**: Tell the kernel to not wait for a write request to complete on this device. The default action is to wait for a write request to complete before returning control to the calling process.
- **C_MGR_IS_MP**: Identify the driver as safe for use in a multiprocessing environment. The 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 also identifies that after the associated buf structure has been marked iodone, physio will remap the buffer to user space.

**SEE ALSO**

driver_close(9E), driver_ioctl(9E), driver_open(9E), driver_psize(9E), driver_read(9E), driver_select(9E), driver_strategy(9E), driver_write(9E), wsio_drv_info(9E), drv_info(9S), physio(9F), select(2)
free_isc(9F)

NAME
free_isc - Free a driver's ISC entry.

SYNOPSIS
#include <sys/wsio.h>
int free_isc(
    struct isc_table_type *isc
);

PARAMETERS
isc Pointer to an ISC entry.

DESCRIPTION
The free_isc kernel function frees an ISC entry that was obtained explicitly by a driver by
using get_new_isc or another similar service. Call free_isc after a severe driver-disabling
error or before a driver is unloaded.

RETURN VALUES
0 Successful completion.
-1 Error

CONSTRAINTS
None

SEE ALSO
get_new_isc(9F)
get_new_isc(9F)

NAME

get_new_isc - 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

None

DESCRIPTION

The get_new_isc WSIO function allocates a new ISC structure when one more is needed, like a multifunction card.

If the isc->ftn_no field is not -1 in an entry for a multifunction card, the driver_attach routine must 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 performs the following:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus_type</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>my_isc</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>if_reg_ptr</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>bus_info</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>ftn_no</td>
<td>Set to -1, the caller must correctly set this field after call</td>
</tr>
<tr>
<td>old_isc-&gt;next_ftn</td>
<td>Set to the new isc</td>
</tr>
<tr>
<td>if_info</td>
<td>Allocated and then copied from old_isc</td>
</tr>
<tr>
<td>new-&gt;next_ftn</td>
<td>Set to NULL</td>
</tr>
<tr>
<td>ifsw</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>if_drv_data</td>
<td>Copied from old_isc</td>
</tr>
<tr>
<td>gfsw</td>
<td>Allocated and copied from old_isc if old_isc-&gt;gfsw is not NULL</td>
</tr>
</tbody>
</table>

RETURN VALUES

<>NULL Success. The value is a pointer to a new ISC structure.

NULL Failure. The get_new_isc routine was unable to allocate memory for the new ISC structure.

CONSTRAINTS

None
install_driver(9F)

NAME

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

SYNOPSIS

#include <gio.h>

int install_driver(
        drive_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.

See the HP-UX 11i v2 Driver Development Guide for information.

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

None

SEE ALSO

wsio_install_driver(9F),
io_hw_path_to_node(9F)

NAME

io_hw_path_to_node - Convert a hardware path to an I/O tree node token.

SYNOPSIS

#include <io/gio.h>

void io_hw_path_to_node(
    void *node,
    hw_path_t *hw_path
);

PARAMETERS

node Pointer to an I/O node name.

hw_path Pointer to a hardware path.

DESCRIPTION

The io_hw_path_to_node WSIO function is used to retrieve the I/O tree node at the path specified in hw_path relative to node. If node is NULL, hw_path specifies the entire hardware path.

RETURN VALUES

The requested node token or NULL, if the specified node does not exist.

CONSTRAINTS

None

SEE ALSO

io_hw_path_to_str(9F), io_node_to_hw_path(9F), io_str_to_hw_path(9F).
NAME

io_hw_path_to_str - Convert a hardware path to a character string.

SYNOPSIS

#include <io/gio.h>
#include <io/ioparams.h>

int io_hw_path_to_str(
    char *str,
    void *node,
    hw_path_t *hw_path
);

PARAMETERS

str On return, pointer to a hardware path character string.
node Pointer to an I/O node name.
hw_path Pointer to a hardware path.

DESCRIPTION

The `io_hw_path_to_str` WSIO function converts the path specified by `hw_path` relative to `node` to the string pointed to by `str`. If `node` is NULL, `hw_path` specifies the entire hardware path. The `str` parameter must have enough space to hold the converted string. The maximum length is specified by `MAX_HW_PATH_STR`. If `str` is NULL, no data is transferred; the length of the string is still returned. The string will contain slash (\/) separators after bus nexus nodes and period (\.) separators after other node types. Therefore, a typical string might look like 8/4/1.2.0, where the first two addresses correspond to bus nexus nodes.

If `io_hw_path_to_str` encounters any aliases on the path from the root node to `hw_path`, it substitutes the alias in place of that segment. For example, for hardware path \{1, 34, 24, 1, 2, 4\} and an alias \{1, 34, 24\} equal to aaabbb, the resulting `hw_path` string is aaabbb/1.2.4.

RETURN VALUES

GIO_ERROR The specified node does not exist.

CONSTRAINTS

None

EXAMPLES

{    hw_path_t hw_path;    char my_string[MAX_HW_PATH_STR];    io_node_to_hw_path(my_node, NULL, &hw_path);    io_hw_path_to_str(my_string, NULL, &hw_path);    printf("%s\n", my_string); }

SEE ALSO

`io_hw_path_to_node(9F), io_node_to_hw_path(9F), io_str_to_hw_path(9F).`
io_mkdev(9F)

NAME

io_mkdev - Converts an I/O tree note and option string into a dev_t.

SYNOPSIS

#include <io/gio.h>

dev_t io_mkdev(
    void *node,
    int dev_type,
    char *options
);

PARAMETERS

node The I/O tree node token for which a dev_t is requested.

dev_type The device type, either D_BLK or D_CHR.

options A string representing the driver options.

DESCRIPTION

The io_mkdev WSIO service creates dev_t's for devices or queries the dev_t of a specified I/O tree node. It returns the dev_t for a specified I/O tree node handle, the device type (D_BLK or D_CHR), and the device-specific options.

RETURN VALUES

If successful, returns the dev_t for a specified I/O node handle.

On error, returns (dev_t)-1.

CONSTRAINTS

None
io_node_to_hw_path(9F)

NAME

io_node_to_hw_path - Convert an I/O tree node token to a hardware path.

SYNOPSIS

#include <io/gio.h>

int io_node_to_hw_path(
    void *node,
    void *base,
    hw_path_t *hw_path
);

PARAMETERS

node Pointer to an I/O node name.
base Pointer to a base I/O node.
hw_path Pointer to a hardware path.

DESCRIPTION

The io_node_to_hw_path WSIO function fills hw_path with the hardware path relative to base of the given node. If base is NULL, hw_path indicates the full hardware path.

RETURN VALUES

GIO_ERROR The node parameter is not a descendent of base.
GIO_SUCCESS The I/O node token was successfully converted.

CONSTRAINTS

None

SEE ALSO

io_hw_path_to_node(9F), io_hw_path_to_str(9F), io_str_to_hw_path(9F).
NAME
io_query - Get information for a specified token.

SYNOPSIS
#include <io/gio.h>

int io_query(
    io_token_t token,
    int type,
    char *key,
    void *ptr
);

PARAMETERS
token      A value previously obtained by a call to io_search.
type       Specifies one of the following search types:
            S_IOTREE   Search the I/O tree based on the given criteria.
            S_BDEVSW   Search the block device switch table.
            S_CDEVSW   Search the character device switch table.
key        Specifies a string corresponding to a defined query field. For a list of keys, see
            io_search(9F).
ptr        Pointer to a buffer containing data returned for the corresponding key.

DESCRIPTION
The io_query WSIO routine gets information about a token previously obtained by a call to
io_search. The data corresponding to key is copied to the buffer pointed to by ptr. The key
parameter must be one of the strings listed in io_search(9F). If ptr is NULL, io_query returns
the number of bytes that would have been copied.

RETURN VALUES
Number of bytes copied
0          The key is undefined for this token.

CONSTRAINTS
None

SEE ALSO
io_search(9F)
NAME

io_search - Search I/O system data structures.

SYNOPSIS

#include <sys/ioparams.h>

void *io_search(
    void *token,
    int type,
    int qual,
    char *key1,
    void *dat1 ...,
    char *keyN,
    void *datN,
    NULL
);

PARAMETERS

token  A token returned by a previous call to io_search, or NULL if this is the first search.

type  Specifies one of the following search types:
    S_IOTREE  Search the I/O tree based on the given criteria.
    S_BDEVSW  Search the block device switch table.
    S_CDEVSW  Search the character device switch table.

qual  Specifies one or more qualifiers for search. Use OR to specify more than one. Most
      qualifiers work for the S_IOTREE searches only. Defined values are:
      Q_SW  Software is associated with an I/O tree node.
      Q_HW  Hardware is associated with an I/O tree node.
      Q_PSEUDO  Is a pseudo device driver. Device switch table searches only.
      Q_DEVSW  Has switch table entry points (for example, LDM).
      Q_NEW  Newly found hardware.
      Q_TRANS  Allows transparent nodes to be returned.
      Q_GIO_INT  Allows GIO internal nodes to be returned (deleted nodes).
      Q_CONSOLE  Matches the console device.
      Q_DUMP  Matches dump devices.
      Q_ROOT  Matches root filesystem devices.
      Q_SWAP  Matches the swap device.
      Q_BOOT  Matches the boot device.
      Q_SAVE_CONF  Matches if the DRV_SAVE_CONF flag is set for the driver associated
                    with the node.
      Q_DYN_MAJOR  Matches if the major can be dynamically allocated. Device switch
                    table searches only.
      Q_VIRTBUS  Matches software controlled buses (virtual buses) and the virtual
                  root node.

key  Specifies a string corresponding to a defined query field.
Specifies a pointer to data to match for the query field indicated in key. There must be a dat for every listed key.

DESCRIPTION

The io_search WSIO function searches for I/O system information. The search ends on an entry in the I/O tree or one of the other tables that can be specified in type. The entry must pass the qualifier test. And the query of all of its keys equal to all of its dats. You can use the token parameter with the io_query function to obtain information about the token. The defined keys are listed in the following table:

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_major</td>
<td>int</td>
<td>Block major number.</td>
</tr>
<tr>
<td>bus_type</td>
<td>char *</td>
<td>The bus type associated with this node.</td>
</tr>
<tr>
<td>c_major</td>
<td>int</td>
<td>Character major number.</td>
</tr>
<tr>
<td>cdio</td>
<td>char *</td>
<td>The name of the associated CDIO.</td>
</tr>
<tr>
<td>child</td>
<td>void *</td>
<td>Token corresponding to a child of the specified token.</td>
</tr>
<tr>
<td>class</td>
<td>char *</td>
<td>A character string representing the class of device.</td>
</tr>
<tr>
<td>def_binding</td>
<td></td>
<td>For deferred binding support.</td>
</tr>
<tr>
<td>deferred_instance</td>
<td></td>
<td>For deferred binding support.</td>
</tr>
<tr>
<td>deferred_name</td>
<td></td>
<td>For deferred binding support.</td>
</tr>
<tr>
<td>description</td>
<td>char *</td>
<td>A character string representing the description of the device.</td>
</tr>
<tr>
<td>driver_name</td>
<td>char *</td>
<td>A character string representing the driver's name or handle.</td>
</tr>
<tr>
<td>health</td>
<td></td>
<td>The health of the specified token.</td>
</tr>
<tr>
<td>hw_path</td>
<td>hw_path_t</td>
<td>The hardware path of the specified node.</td>
</tr>
<tr>
<td>id_bytes</td>
<td>void *</td>
<td>The ID bytes of the specified node.</td>
</tr>
<tr>
<td>instance</td>
<td>int</td>
<td>The instance number of the specified node.</td>
</tr>
<tr>
<td>io_flags</td>
<td>int</td>
<td>Returns the io_flags of the given node.</td>
</tr>
<tr>
<td>is_block</td>
<td>int</td>
<td>1, if the driver can be accessed through the block switch table; else, 0.</td>
</tr>
<tr>
<td>is_char</td>
<td>int</td>
<td>1, if the driver can be accessed through the character switch table; else, 0.</td>
</tr>
<tr>
<td>is_pseudo</td>
<td>int</td>
<td>1, if the token corresponds to a pseudo device; else, 0.</td>
</tr>
<tr>
<td>is_remote</td>
<td>int</td>
<td>1, if the token corresponds to a remote device; else, 0.</td>
</tr>
<tr>
<td>is_virtbus</td>
<td>int</td>
<td>1, if the token corresponds to a software virtual bus; else, 0.</td>
</tr>
<tr>
<td>minor_num</td>
<td>int</td>
<td>A device number corresponding to the specified token is returned. If anything other than the default minor number is needed, use io_mkdev.</td>
</tr>
</tbody>
</table>
### Key Name | Data Type | Description
--- | --- | ---
name | char * | The name of the specified token. By convention, this is equivalent to the driver name.
parent | void * | Token corresponding to the parent of the specified token.
proc_core | | Returns the processor core information.
proc_socket | | Returns the processor socket information.
proc_thread | | Returns the processor thread information.
registers | unsigned int | Returns an array of pointers representing the register sets for the specified token. If the module has two register sets, an integer array with two elements is returned. For PA modules, element 0 is the HPA and element 1 is the SPA.
registers64 | unsigned long | The 64-bit version of registers.
save_conf | int | 1 if this node is to be saved in the ioconfig file; else, 0.
sibling | void * | Token corresponding to the next sibling of the specified token.
state | char * | A character string corresponding to the state (for example, CLAIMED and UNCLAIMED) of the indicated node.
subtree | hw_path | When used in io_query, this query behaves like hw_path. When used in io_search, the search passes if the entry is in the subtree of the dat passed in.
type | char * | A character string corresponding to the specified node's type. For example, TDEVICE and T_MEMORY.
visibility | hw_path | When used in io_query, this query behaves like hw_path. When used in io_search, the search passes if the entry is visible from the dat passed in. The definition of visibility depends on the device's subsystem. In general, if interfaces can perform I/O to a device, the device is visible.

The following table lists the health attributes and their values:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>String Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEALTH_ONLINE</td>
<td>online</td>
<td>The I/O Node is online and functional. For example, LUN and lunpath are online and ready to issue I/Os.</td>
</tr>
<tr>
<td>HEALTH_OFFLINE</td>
<td>offline</td>
<td>The I/O Node has gone offline. For example, lunpath has gone offline; HBA Controller has gone link down; tgtpath has gone offline; and all paths to the LUN have gone offline causing the LUN to be inaccessible.</td>
</tr>
<tr>
<td>HEALTH_LIMITED</td>
<td>limited</td>
<td>The I/O Node is operating in a performance-degraded (suboptimal) state. For example, an aggregate group (LUN or an iSCSI session) has one of its links offline. Either one or more lunpaths to a LUN are offline, one of more connections in an iSCSI session are offline, or one or more 32 PHYs in a SAS wide port are offline.</td>
</tr>
<tr>
<td>HEALTH_BROKEN</td>
<td>broken</td>
<td>The I/O Node has detected an error condition that needs manual administrator intervention. For example, lunpath has an authentication failure on LUN WWID; tgtpath has an authentication failure on Port WWN; and HBA Controller has detected an unrecoverable PCI Error.</td>
</tr>
<tr>
<td>HEALTH_TESTING</td>
<td>testing</td>
<td>The I/O Node is in a diagnostic mode and is being subject to Testing operations. For example, HBA Controller is in a loopback mode; LUN or lunpath is being subject to diagnostic testing.</td>
</tr>
</tbody>
</table>
### RETURN VALUES

If successful, `io_search` returns a token for use with later calls to `io_search` or `io_query`. If no match found, the function returns NULL.

### CONSTRAINTS

None

### EXAMPLES

The following example searches for all disks in the system and prints the hardware path:

```c
token = NULL;
do {
    hw_path_t hw_path;
    char hw_path_s[MAX_HW_PATH];
    token = io_search(token, S_IOTREE, Q_HW, "class", "disk", NULL);
    if (token != NULL) {
        io_query(token, S_IOTREE, "hw path", &hw_path);
        io_hw_path_to_str(hw_path_s, NULL, &hw_path);
        printf("%s\n", hw_path_s);
    }
} while (token != NULL);
```

The following example prints the names of all character pseudo drivers in the system:

```c
token = NULL;
do {
    char pseudo_name[MAX_NAME_LEN];
    token = io_search(token, S_CDEVSW, Q_PSEUDO, NULL);
    if (token != NULL) {
        io_query(token, S_CDEVSW, "driver_name", pseudo_name);
        printf("driver name = %s\n", pseudo_name);
    }
} while (token != NULL);
```

The following example prints the character major number and instance number of all tapes in the system that are SIO-style drivers:

```c
*token = NULL;
key[0] = "class"; ptr[0] = (void *)"tape";
key[1] = "cdio"; ptr[1] = (void *)"SIO";
do {
    int c_major, instance;
    token = io_search_array(token, S_IOTREE, 0, key, ptr);
    if (token != NULL) {
        io_query(token, S_IOTREE, "c_major", &c_major);
        io_query(token, S_IOTREE, "instance", &instance);
    }
} while (token != NULL);
```
printf("cmajor=%d instance=%d\n", c_major, instance);
} while (token != NULL);

SEE ALSO

io_query(9F)
io_str_to_hw_path(9F)

NAME

io_str_to_hw_path - Convert a character string to a hardware path.

SYNOPSIS

#include <io/gio.h>

int io_str_to_hw_path(
    char *str,
    void *node,
    hw_path_t *hw_path
); 

PARAMETERS

str Pointer to a hardware path character string.
node Pointer to an I/O node name.
hw_path Pointer to a hardware path.

DESCRIPTION

The io_str_to_hw_path WSIO function converts a string representing the partial path from node and returns it in hw_path with the hardware path relative to node. If node is NULL, hw_path indicates the full hardware path.

The hw_path parameter receives the full hardware path; however, its first_index points to where the relative hw_path starts.

If str contains an invalid hardware path, io_str_to_hw_path completes as long as it is possible to do so. If str contains a non-existent alias reference or a number is too large to be in one address level, the function fails.

RETURN VALUES

GIO_ERROR The string does not represent a valid path and it is impossible to create one.
GIO_SUCCESS The string was successfully converted.

CONSTRAINTS

None

EXAMPLES

char *str_path=“es/456fff780787c56a”;
char *str_path2=“4/5.4.0”;
hw_path_t hw_path;
hw_path_t hw_path_2;

calculate_path() {
    io_str_to_hw_path(str_path,NULL,&hw_path);
    io_str_to_hw_path(str_path2,NULL,&hw_path2);
}

SEE ALSO

io_hw_path_to_node(9F), io_hw_path_to_str(9F), io_node_to_hw_path(9F).
iodone(9F)

NAME

iodone - 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 must call biodone directly instead of calling iodone.

The biodone kernel function completes the buffer I/O transaction. There must 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

Do not call while holding a spinlock of order >= BUF_HASH_LOCK_ORDER.

WARNINGS

The biodone function calls panic if B_DONE is set in bp->b_flags upon entry.

SEE ALSO

iodone(9F), biowait(9F), iowait(9F)
NAME
iowait – 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 must
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    Successful completion.
<>0   Error

CONSTRAINTS
Do not call in an interrupt context.
Do not call while holding a spinlock.

EXAMPLES
int error;
struct buf *bp;
error = iowait(bp);

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

SEE ALSO
biodone(9F), biowait(9F), geterror(9F), iodone(9F)
isc_claim(9F)

NAME

isc_claim - Marks an ISC entry as claimed by the driver.

SYNOPSIS

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

void isc_claim(
    struct isc_table_type *isc,
    wsio_drv_info_t drv_info
);

PARAMETERS

isc Pointer to the ISC entry associated with an interface card or device.

drv_info Pointer to the wsio_drv_info_t structure of the driver that is claiming the ISC structure.

DESCRIPTION

The isc_claim WSIO function marks an ISC entry as claimed by the driver. The isc_claim function 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 must 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

None

EXAMPLE

static int
mydrv_attach(uint32_t id, struct isc_table_type *isc)
{
    ...
    if (id == MY_DEVICE_HW_ID) {
        /*
        * Specify the interface init function that is
        * called for each claimed ISC entry after the
        * attach chain processing has completed.
        */
        isc->gfsw->init = mydrv_if_init;
        
        /*
        * Claim the ISC entry representing the device.
        */
        isc_claim(isc, &mydrv_info);
    }
    
    /*
    * Call the next driver on the attach chain.
    */
    return (*mydrv_saved_attach)(id, isc);
}
SEE ALSO

*driver_attach*(9E)
isc_table_type(9S)

NAME
isc_table_type - 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.

I/O Switch Tables
The I/O system supports two I/O interface switch tables through fields in the ISC structure. The isc->gfsw is intended for use by the system. The 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
Table 2-2: “Driver Relevant Structure Fields” shows a list of driver accessible fields in the ISC structure that are initialized by the system.

Table 2-2 Driver Relevant Structure Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus_type</td>
<td>int</td>
</tr>
<tr>
<td>if_id</td>
<td>int</td>
</tr>
<tr>
<td>if_info</td>
<td>caddr_t</td>
</tr>
<tr>
<td>if_info-&gt;flags</td>
<td>int</td>
</tr>
<tr>
<td>if_reg_ptr</td>
<td>caddr_t</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 drivers 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 the interface driver’s attach routine is called, if the driver claims the card you can use this field as desired. For example, if the PCI driver memory space is not mapped due to size constraints, call **map_mem_to_host** and store the returned virtual address from that call in this field.

Table 2-3 “Driver-Initialized ISC Fields” shows a list of fields in the ISC structure that are initialized by drivers.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>card_ptr</td>
<td>volatile int *</td>
</tr>
<tr>
<td>gfsw-&gt;init</td>
<td>int (*)(struct isc_table_type *)</td>
</tr>
<tr>
<td>if_drv_data</td>
<td>caddr_t</td>
</tr>
<tr>
<td>if_isc</td>
<td>void *</td>
</tr>
<tr>
<td>ifsw</td>
<td>caddr_t</td>
</tr>
<tr>
<td>my_address</td>
<td>char</td>
</tr>
<tr>
<td>owner</td>
<td>struct buf *</td>
</tr>
<tr>
<td>state</td>
<td>unsigned int</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. |
**m_wsio_funcnum(9F)**

**NAME**
m_wsio_funcnum - 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**
None

**CONSTRAINTS**
None

**SEE ALSO**
m_wsio_selcode(9F)

NAME
m_wsio_selcode - 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
None

CONSTRAINTS
None

SEE ALSO
NAME
m_wsio_vsc - Return the system bus module number for a device number.

SYNOPSIS
#include <sys/wsio.h>
int m_wsio_vsc(
    dev_t dev,
    wsio_drv_info_t *drv_hdr_ptr
);

PARAMETERS
dev The dev_t number of a device.
drv_hdr_ptr Pointer to the wsio_drv_info_t structure for the device.

DESCRIPTION
The m_wsio_vsc WSIO function returns the system bus module number for device number dev.

RETURN VALUES
None

CONSTRAINTS
None

SEE ALSO
NAME
mod_wsio_attach_list_add - Add the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS
#include <sys/wsio.h>
int mod_wsio_attach_list_add(
    int type,
    void *attach_func
);

PARAMETERS

type Type of WSIO attach list.
attach_func Pointer to the driver attach function.

DESCRIPTION
The mod_wsio_attach_list_add WSIO function adds the driver attach function pointer attach_func to the WSIO attach list specified by type.
Dynamically loadable drivers call mod_wsio_attach_list_add when they are loaded into the kernel.
The type parameter specifies the attach list to use. Valid values are:
MOD_WSIO_CORE For Core I/O attach list
MOD_WSIO_EISA For EISA I/O attach list
MOD_WSIO_PCI For PCI I/O attach list
The attach_func parameter points to the driver attach function that will be called by the WSIO environment to see if the driver wants to claim a device.

RETURN VALUES
0 Successful completion.
1 Error

CONSTRAINTS
None

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(9F)
mod_wsio_attach_list_remove(9F)

NAME

mod_wsio_attach_list_remove - Remove the driver attach function pointer to the specified WSIO attach list.

SYNOPSIS

#include <sys/wsio.h>

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
ENOMATCH The attach_func parameter does not match the value passed to mod_wsio_attach_list_add.

CONSTRAINTS

None

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(9F)
modlink(9S)

NAME

modlink - Structure contains information about the operations supported by the driver and information about the driver typespecific data.

SYNOPSIS

#include <sys/moddefs.h>

DESCRIPTION

A pointer to the modlink structure is stored in the modwrapper structure. The modlink structure gives information about the operations supported by the driver and the driver typespecific information. The exact contents are dependent on the type of the driver.

STRUCTURE MEMBERS

The modlink structure is defined as follows:

```c
struct modlink {
    struct mod_operations *ml_ops;
    void *ml_type_data;
};
```

The elements of the modlink structure are shown in Table 2-4 “Modlink Structure”.

Table 2-4 Modlink Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml_ops</td>
<td>A pointer to a mod_operations switch table for the module type. The structure depends on the type of DLKM driver module being defined.</td>
</tr>
<tr>
<td>ml_type_data</td>
<td>Pointer to the mod_type_data structure. The mod_type_data contains a module description string and the second field is set to NULL.</td>
</tr>
</tbody>
</table>

EXAMPLE

The name of the modlink structure array must be driver_mod_link. The array of the modlink structure must be terminated by a {NULL, NULL} entry.

The following is an example of a modlink structure array:

```c
static struct mod_type_data driver_drv_link = {
    "Driver description string",
    (void *)NULL
};
struct modlink driver_mod_link[] = {
    {&gio_mod_ops, &driver_drv_link },
    {NULL,      NULL      }
};
```

SEE ALSO

driver_load(9E), driver_unload(9E), modwrapper(9S)
NAME

modwrapper - Structure to specify DLKM driver entry points and configuration information.

SYNOPSIS

#include <sys/moddefs.h>

DESCRIPTION

The modwrapper structure must be named as drivername_wrapper, and must be externally visible.

STRUCTURE MEMBERS

The modwrapper structure is defined as follows:

extern struct mod_conf_data drivername_conf_data;

struct modwrapper {
    int mw_rev;
    int (*mw_load)(void *);
    int (*mw_unload)(void *);
    void (*mw_halt)(void);
    void *mw_conf_data;
    struct modlink *mw_modlink;
};

The fields of the modwrapper structure are shown in Table 2-5 "Modwrapper Structure".

Table 2-5 Modwrapper Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mw_rev</td>
<td>Driver revision number. The field must always be set to MODREV. MODREV is defined in &lt;sys/moddefs.h&gt;.</td>
</tr>
<tr>
<td>mw_load</td>
<td>Pointer to the driver's load function.</td>
</tr>
<tr>
<td>mw_unload</td>
<td>Pointer to the driver's unload function.</td>
</tr>
<tr>
<td>mw_halt</td>
<td>Reserved for future use. Must be set to NULL.</td>
</tr>
<tr>
<td>mw_conf_data</td>
<td>Pointer to driver's configuration data. This field must always be set to drivername_conf_data. The data structure is automatically generated as part of the driver's metadata.</td>
</tr>
<tr>
<td>mw_modlink</td>
<td>Pointer to an array of structmodlink.</td>
</tr>
</tbody>
</table>

NOTE

The source file that defines the modwrapper structure will need to have an extern for mod_conf_data as drivername_conf_data.

SEE ALSO

driver_load(9E), driver_unload(9E), modlink(9S)
wsio_activate_probe(9F)

NAME

wsio_activate_probe - Activate the probe function for a driver.

SYNOPSIS

void wsio_activate_probe(
    char *probe_name,
    struct drv_info *drv_infop
);

PARAMETERS

probe_name Name of the device probe function as registered by
            wsio_register_dev_probe

drv_infop Pointer to the driver drv_info structure

DESCRIPTION

The wsio_activate_probe WSIO function connects the probe function for a dynamically
loadable interface driver to the driver drv_info structure. The wsio_activate_probe function
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

None

EXAMPLES

static wsio_drv_info_t mydrv_info = { ... };

int mydrv_load(void * arg)
{
    /*
    * Use the drv_info passed to 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(9F), wsio_unregister_dev_probe(9F)
NAME

wsio_alloc_mem - Allocate memory for DMA buffers or control structures.

SYNOPSIS

#include <sys/wsio.h>

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 Handle allocated by a call to wsio_alloc_mem_handle.
size Size of the buffer in bytes.
vaddr Address of the pointer to the allocated buffer. Pointer is set to NULL if unable
to allocate a buffer.
flags Flags which describe restrictions:
    WSIO_SLEEP_OK Flag to indicate service can sleep if waiting for resources.

DESCRIPTION

This WSIO service allocates memory used for DMA buffers or control structures. The first
parameter to the service must be a mem_handle allocated by a call to wsio_alloc_mem_handle.

RETURN VALUES

WSIO_ALLOC_OK The buffer was allocated.
WSIO_ALLOC_OUT_OF_RESOURCES Unable to allocate the specified resources.

CONSTRAINTS

If WSIO_SLEEP_OK is set in flags:
  • Do not call while holding a spinlock
  • Do not call in interrupt context.

EXAMPLES

wsio_alloc_status_t status;
wsio_vaddr_t vaddr;
wsio_mem_handle_t mem_handle;
size_t size;

if(wsio_alloc_mem_handle(my_isc, &mem_handle,
    WSIO_32BIT_MEMORY |
    WSIO_IO_CONTIGUOUS) == WSIO_ALLOC_OK)
{
    /* Allocate memory with a non-blocking call
       to wsio_alloc_mem() */
    status = wsio_alloc_mem(mem_handle, size, &vaddr, 0);
}

SEE ALSO

wsio_alloc_mem_handle(9F), wsio_free_mem(9F), wsio_free_mem_handle(9F)
wsio_alloc_mem_handle(9F)

NAME
wsio_alloc_mem_handle -- Specify the type of memory to allocate.

SYNOPSIS
#include <sys/wsio.h>

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 Pointer to the device isc structure.
mem_handle Pointer to the returned handle.
attribs Attributes describing the criteria for the type of memory to allocate.
    WSIO_OPTIMIZE_FOR_DEVICE Allocate memory close to the device. On Multicell systems, memory is allocated on the same cell as the device.
    WSIO_OPTIMIZE_FOR_CPU Allocate memory close to the current CPU. On Multicell systems, memory is allocated on the same cell as the CPU. This is the default behavior.
    WSIO_32BIT_MEMORY Allocate memory below 4GB.
    WSIO_IO_CONTIGUOUS On platforms without an I/O PDIR, physically contiguous memory is allocated.
    WSIO_ALIGN_ON_SIZE Allocate memory aligned to a power-of-two value greater than or equal to the requested size.

DESCRIPTION
Drivers call this service to specify the type of memory to allocate. The service returns a mem_handle, which is passed to 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, HP recommends 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 The handle was allocated.
WSIO_ALLOC_OUT_OF_RESOURCES Unable to allocate the specified resources.
WSIO_INVAL_PARAM A parameter was not valid.
CONTRAINTS

Do not call in interrupt context.
Do not call while holding a spinlock.

EXAMPLE

wsio_alloc_status_t status;
wsio_vaddr_t vaddr;
wsio_mem_handle_t mem_handle;
size_t size;

if(wsio_alloc_mem_handle(my_isc, &mem_handle,
    WSIO_32BIT_MEMORY |
    WSIO_IO_CONTIGUOUS) == WSIO_ALLOC_OK)
{
    /* Allocate memory with a non-blocking call
       to wsio_alloc_mem() */
    status = wsio_alloc_mem(mem_handle, size, &vaddr, 0);
}

SEE ALSO

wsio_alloc_mem(9F), wsio_free_mem(9F), wsio_free_mem_handle(9F)
NAME
wsio_allocate_dma_handle - Obtain a handle used to setup DMA.

SYNOPSIS
#include <sys/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 could not be allocated.

CONSTRAINTS
None

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(9F), wsio_dma_pass_thru(9F), wsio_fastmap_dma_buffer(9F),
wsio_free_dma_handle(9F), wsio_free_shared_mem(9F), wsio_flush_shared_mem(9F),
wsio_init_map_context(9F), wsio_iqva_to_phys(9F), wsio_map_dma_buffer(9F),
wsioremap_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F),
wsiounmap_dma_buffer(9F)
wsio_allocate_shared_mem(9F)

NAME

wsio_allocate_shared_mem - Set up an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <sys/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
);

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 allocate.
iova Pointer that contains the I/O virtual address upon completion. A
wsio_iova_t variable must be allocated by the driver, and the pointer
to this is what must be passed into the macro.
vaddr Pointer that contains the host virtual address upon completion. A
wsio_vaddr_t variable must be allocated by the driver, and the pointer
to this is what must be passed into the macro.
shared_mem_attr 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 WSIO_IO_SHMEM_OPTIMIZE_DEVICE_LATENCY is used. The behavior of the allocation is also affected by attributes set using
wsio_dma_set_device_attributes, and
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>WSIO_IO_SHMEM_OPTIMIZE_DEVICE_LATENCY</td>
<td>Allocation must optimize for device access latency. If possible, allocate object in memory local to a bus bridge.</td>
</tr>
<tr>
<td>WSIO_IO_SHMEM_OPTIMIZE_HOST_LATENCY</td>
<td>Allocation must optimize for host access latency. If possible, allocate in host memory.</td>
</tr>
<tr>
<td>WSIO_IO_SHMEM_DMA_ALLOC_COMPATIBLE</td>
<td>Behave exactly as the 10.X dma_alloc service. This type is for compatibility with 10.X only.</td>
</tr>
<tr>
<td>WSIO_IO_SHMEM_INBOUND</td>
<td>This attribute can be ORed with the other attributes to indicate the buffer is used exclusively for inbound DMA.</td>
</tr>
<tr>
<td>WSIO_IO_SHMEM_OUTBOUND</td>
<td>This attribute can be ORed with the other attributes to indicate the buffer is used exclusively for outbound DMA.</td>
</tr>
</tbody>
</table>
DESCRIPTION

The `wsio_allocate_shared_mem` 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, `wsio_map_dma_buffer` must be used.

If a callback function is set up (see `wsio_set_dma_callback`), and no resources are available when the call is made, `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

- **WSIO_MAP_OK**: Success
- **WSIO_MAP_W_CALLBACK**: Returned if no resources are available and a callback is registered.
- **WSIO_MAP_E_NO_RESOURCES**: Returned if no resources are available and no callback is registered.
- **WSIO_MAP_E_RESOURCE_ERROR**: Returned if cannot allocate resources. If this is returned, the allocation will never succeed.
- **WSIO_MAP_E_PARAMETER_ERROR**: Returned on bad parameter (Software bug).
- **WSIO_MAP_E_UNKNOWN_ERROR**: Returned if there is an unknown error.

CONSTRAINTS

None

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(9F), wsio_dma_pass_thru(9F), wsio_free_dma_handle(9F),
wsio_free_shared_mem(9F), wsio_flush_shared_mem(9F), wsio_iova_to_phys(9F),
wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F)
WSIO_BIG_ENDIAN(9F)

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

SYNOPSIS
#include <sys/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 must be performed. Endian swapping might be necessary for any data transfers between the I/O bus and local host memory.
The WSIO_BIG_ENDIAN macro can be called in a non-blocking context.

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

CONSTRAINTS
None

EXAMPLE
if (WSIO_BIG_ENDIAN(isc_entry)) {
    /* No endian swapping necessary */
} else {
    /* Endian swapping must be performed */
}

SEE ALSO
WSIO_LITTLE_ENDIAN(9F)
wsio_cfg_inXX(9F)

NAME

wsio_cfg_inXX - Macros to read from configuration space.

SYNOPSIS

#include <sys/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

None

EXAMPLE

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(9F), wsio_map_cfg_handle(9F), wsio_unmap_cfg(9F)
NAME

wsio_cfg_outXX - Macros for writing to configuration space.

SYNOPSIS

#include <sys/wsio.h>

void wsio_cfg_outXX(
    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 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_outXX macros are called by device drivers to write to configuration space. The cfg_handle and the 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

None

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(9F), wsio_map_cfg_handle(9F), wsio_unmap cfg(9F)
wsio_create_attribute(9F)

NAME

wsio_create_attribute - Registers a new attribute with an interface.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_create_attribute(
    IN struct isc_table_type isc,
    IN char *name,
    IN uintptr_t *value,
    IN size_t size,
    IN wsio_attrib_flags_t flags
);

PARAMETERS

isc   The isc handle of the interface that the attribute will be is associated with.
flags Flags indicating behavior of attribute.
name  A character string representing the name of the attribute.
value A pointer to the attributes current data.
size  The size of the data in bytes.

DESCRIPTION

The service is called to create a new attribute for an interface. The isc parameter identifies the interface. The second parameter is the name of the new attribute. The value, size, and flags parameters identify the initial data for the attribute. The flags parameter identifies characteristics of the attribute and the data referenced by value. This service is safe to call on the ICS unless the WSIO_WAIT_OK flag is specified in the flags parameters.

The WSIO_ATTR_EXPORT flag indicates that this attribute will be visible to any children.

wsio_attrib_flags_t Description
WSIO_COPYDATA If set, value is assumed to contain an address that references a data buffer and the contents of the buffer is copied; otherwise, value is assumed to contain the immediate data that is saved.
WSIO_WAIT_OK If resources are not available the call will block until they are.
WSIO_ATTR_EXPORT The attribute will be exported to any children.

When creating an attribute a reference to a kernel memory data structure can be saved by simply passing in the address and size of the structure as the value and size parameters. The kernel memory data structure must then be persistent in memory as long as the attribute exists. If the caller wishes to save a copy of a structure then they must set the WSIO_COPYDATA flag. The service will then copy the contents of the data to an internal buffer.

RETURN VALUES

WSIO_OK The attribute is successfully created.
WSIO_ERROR An error occurred.
WSIO_ATTR_EXIST The attribute exists.
WSIO_NO_RESOURCE Resources are not available and WSIO_WAIT_OK was not set.

CONSTRAINTS

Cannot be called on ICS.
SEE ALSO

wsio_modify_attribute(9F), wsio_get_attribute(9F), wsio_destroy_attribute(9F), wsio_sizeof_attribute(9F).
wsio_create_interface(9F)

NAME

wsio_create_interface - Register a new interface with the WSIO.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_create_interface(  
  IN struct isc_table_type *parent,  
  IN hw_path_t *path,  
  IN wsio_mod_type_t type,  
  IN char *id,  
  IN char *name,  
  IN char *desc,  
  IN char *drvname,  
  OUT struct isc_table_type **isc  
);

PARAMETERS

- **parent** The isc handle of a parent of the new interface if path is relative; else, NULL.
- **path** Hardware path of new interface.
- **type** WSIO_INTERFACE or WSIO_TRANS.
- **id** ID string of new module.
- **name** Name string of new module.
- **desc** Description string of new module.
- **drvname** Reserved for future use.

DESCRIPTION

This service is called by drivers to create an I/O interface. It can be called in a drivers install, probe, or scan routine.

The types of interfaces that can be created are:

- WSIO_INTERFACE — An interface.
- WSIO_TRANS — A transparent interface.

WSIO_TRANS is a specialized type of interface. It has no associated hardware, and is used to create hardware path elements. Both types will have an isc handle associated with them and must be created in the drivers scan or probe routine. The isc handle can be passed to other WSIO services.

The path and parent parameters are used together to determine the hardware path of the new interface. If the parent parameter is not NULL, path is assumed to be relative to the parent; otherwise, it is assumed to be absolute.

The service first checks to see if the interface already exists at the specified hardware path. If it does not, it creates it; otherwise, it compares the id, name, and desc attributes of the existing interface with those passed in as parameters. If they are different, it updates the id, name, and desc attributes with the new values and reports the difference to the I/O subsystem.

The service returns an isc handle for the newly created entry.
NOTE: If this function is used to create a virtual I/O interface (an interface without physical hardware), do not call any functions that use the returned isc handle to access the interface's registers, ports, or configuration space.

RETURN VALUES
The isc handle for the new interface if successful; else, NULL.

CONSTRAINTS
Cannot be called on ICS.

SEE ALSO
wsio_destroy_interface(9F)
**NAME**

wsio_destroy_attribute - Destroy an attribute registered with an interface.

**SYNOPSIS**

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

wsio_ret_code_t wsio_destroy_attribute(
    struct isc_table_type *isc,
    char *name
);
```

**PARAMETERS**

*isc*  
The *isc* handle of the interface with which the attribute is associated.

*name*  
A character string representing the name of the attribute.

**DESCRIPTION**

This service is called to destroy an attribute associated with an interface.

**RETURN VALUES**

- **WSIO_OK**  
The attribute was successfully destroyed.
- **WSIO_ERROR**  
Invalid *isc* structure or attribute name.

**CONSTRAINTS**

Cannot be called on ICS.

**SEE ALSO**

wsio_create_attribute(9F), wsio_modify_attribute(9F), wsio_get_attribute(9F), wsio_sizeof_attribute(9F).
NAME

wsio_destroy_interface - Unregisters an interface with the WSIO.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_destroy_interface(
    struct isc_table_type *isc
);

PARAMETERS

isc      The isc handle associated with the interface.

DESCRIPTION

This service is called to destroy an interface that was create via a call to
wsio_io_create_interface. If the interface has any children they will be implicitly destroyed.

RETURN VALUES

WSIO_OK    The attribute was successfully destroyed.
WSIO_ERROR Invalid isc structure or attribute name.

CONSTRAINTS

Cannot be called on ICS.

SEE ALSO

wsio_create_interface(9F)
NAME

wsio_dma_pass_thru - Call a DMA pass-thru function that might not otherwise be accessible.

SYNOPSIS

#include <sys/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</td>
</tr>
<tr>
<td></td>
<td>processors can access host memory atomically, and can be used for</td>
</tr>
<tr>
<td></td>
<td>synchronization. The parameter's pass_thru_param indicates whether</td>
</tr>
<tr>
<td></td>
<td>shared memory is to be locked (1) or unlocked (0). Implementation of</td>
</tr>
<tr>
<td></td>
<td>this function is not required, so the return code is zero (0) if the</td>
</tr>
<tr>
<td></td>
<td>function is implemented or non-zero if not implemented.</td>
</tr>
<tr>
<td>WSIO_MAP_PT_SYNC_BUS</td>
<td>Causes any FIFOs, buffers, or I/O caches associated with a device to</td>
</tr>
<tr>
<td></td>
<td>be synchronized with memory. The parameter must be zero (0). This</td>
</tr>
<tr>
<td></td>
<td>function returns zero (0) if the function is implemented and non-zero if</td>
</tr>
<tr>
<td></td>
<td>it is not.</td>
</tr>
</tbody>
</table>

pass_thru_param Parameter that will get passed into the pass-thru function as an argument.

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

None
EXAMPLE

```c
if (wsio_dma_pass_thru(isc_entry, dma_handle,
       WSIO_MAP_PT_SYNC_BUSSES, 0) != 0) {
    /* The function is not 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(9F)`, `wsio_allocate_shared_mem(9F)`, `wsio_fastmap_dma_buffer(9F)`, `wsio_flush_shared_mem(9F)`, `wsio_free_dma_handle(9F)`, `wsio_free_shared_mem(9F)`, `wsio_init_map_context(9F)`, `wsio_iova_to_phys(9F)`, `wsio_map_dma_buffer(9F)`, `wsio_remap_dma_buffer(9F)`, `wsio_set_device_attributes(9F)`, `wsio_set_dma_attributes(9F)`, `wsio_unmap_dma_buffer(9F)`
wsio_dma_set_device_attributes(9F)

NAME

wsio_dma_set_device_attributes - Associate DMA hints with a device.

SYNOPSIS

#include <sys/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 are:
            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 Dependent
            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 must be flushed from any intermediate buffers as soon as it is referenced. This inhibits any coalescing of data by bus bridges.
            Default value: 0
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_DMA_ATTR_IGN_ALIGN</td>
<td>Specifies the mapping service must not handle cacheline fragments in a special way.</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_INTERLEAVE</td>
<td>IOVA allocation model</td>
</tr>
<tr>
<td></td>
<td>0 DMA streams are normally interleaved (mass-storage).</td>
</tr>
<tr>
<td></td>
<td>1 DMA streams are normally not interleaved (networking).</td>
</tr>
<tr>
<td></td>
<td>2 DMA buffers are static and accessed randomly (low fat).</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_PREFETCH</td>
<td>Specifies how aggressively hardware is to prefetch for outbound DMA.</td>
</tr>
<tr>
<td></td>
<td>0 No prefetch</td>
</tr>
<tr>
<td></td>
<td>1 Moderate prefetch</td>
</tr>
<tr>
<td></td>
<td>2 Aggressive prefetch</td>
</tr>
<tr>
<td></td>
<td>Default value: 1</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_SAFE</td>
<td>Specifies the use of the most conservative coherency model for inbound DMA. Inhibits semicoherent transactions such as WRITE_PURGE unless it is guaranteed that no data in processor caches will be lost.</td>
</tr>
<tr>
<td></td>
<td>1 On</td>
</tr>
<tr>
<td></td>
<td>2 Off</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_TXN_SIZE</td>
<td>Specifies the default transaction size used by the device. This is used by hardware to optimize conversion of transactions between buses.</td>
</tr>
<tr>
<td></td>
<td>Default value: Hardware dependent</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_INBOUND</td>
<td>DMA buffers will be used exclusively for inbound DMA.</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_OUTBOUND</td>
<td>DMA buffers will be used exclusively for outbound DMA.</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
<tr>
<td>WSIO_DMA_ATTR_STABLE</td>
<td>Indicates the buffer will not be modified by another entity while mapped for DMA. This is normally true (1) for data buffers, and false (0) for control structures.</td>
</tr>
<tr>
<td></td>
<td>Default value: 0</td>
</tr>
</tbody>
</table>
param Information dependent on the hint or attribute being set. Check the attribute list for more information.

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`.

The `wsio_dma_set_device_attributes` function 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.

CONSTRAINTS

None

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

`wsioAllocate_dma_handle`(9F), `wsioAllocate_shared_mem`(9F), `wsio_dma_pass_thru`(9F),
`wsioFastmap_dma_buffer`(9F), `wsio_free_dma_handle`(9F), `wsio_free_shared_mem`(9F),
`wsioFlush_shared_mem`(9F), `wsio_init_map_context`(9F), `wsio_iova_to_phys`(9F),
`wsio_map_dma_buffer`(9F), `wsio_remap_dma_buffer`(9F), `wsio_set_dma_attributes`(9F),
`wsio_unmap_dma_buffer`(9F)
wsio_drv_data(9S)

NAME

wsio_drv_data - Driver-specific fields for WSIO drivers.

SYNOPSIS

#include <sys/wsio.h>

DESCRIPTION

The wsio_drv_data_t WSIO structure type, defined in <sys/wsio.h>, contains driver-specific fields for WSIO drivers.

STRUCTURE MEMBERS

Field Purpose

drv_path Follow these guidelines:

For device drivers, drv_path is typically a string that contain the interface card's type and the device's class. For example, scsi_disk.

For interface drivers, drv_path must match the card's type. For example, scsi.

For pseudo drivers, drv_path must match the card's class. For example, graphics.

drv_type One of the following values:

T_INTERFACE The driver controls an interface card.

T_DEVICE The driver controls a hardware device.

drv_flags One of the following values:

DRV_CONVERGED The driver meets the HP-UX Release 10.0 Converged I/O specifications. All new drivers must meet these specifications.

NOT_CONVERGED The driver conforms to the pre-Release 10.0 unconverted specifications.

drv_minor_build Pointer to your minor number formatter. Use NULL if you do not provide one.

drv_minor_decode Pointer to your minor number interpreter. Use NULL if you do not provide one.

EXAMPLES

static wsio_drv_data_t sdisk_data = {
    "scsi_disk",
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};
wsio_drv_info(9S)

NAME

wsio_drv_info - Structure containing pointers to other CDIO and WSIO data structures.

SYNOPSIS

#include <sys/wsio.h>

DESCRIPTION

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

STRUCTURE MEMBERS

drv_info Pointer to a drv_info_t CDIO structure.
drv_ops Pointer to a drv_ops_t CDIO structure.
drv_data Pointer to a wsio_drv_data_t structure.
driver_version Set to WSIO_DRV_CURRENT_VERSION.

SEE ALSO

drv_info(9S), drv_ops(9S), wsio_drv_data_t(9S)
NAME

wsio_fastmap_dma_buffer - Function to map an existing memory object for packet DMA.

SYNOPSIS

#include <sys/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
None

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(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_free_dma_handle(9F), wsio_free_shared_mem(9F), wsio_flush_shared_mem(9F),
wsio_init_map_context(9F), wsio_iova_to_phys(9F), wsio_map_dma_buffer(9F),
wsio_remap_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F),
wsio_unmap_dma_buffer(9F)
wsio_flush_shared_mem(9F)

NAME

wsio_flush_shared_mem - Flush an I/O virtually contiguous DMA buffer.

SYNOPSIS

#include <sys/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 wsio_allocate_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 wsio_allocate_shared_mem. Any non-coherent buffers associated with the memory object are flushed. All parameters to the function must be the same as those passed to the call that allocated the memory.

The 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.

CONSTRAINTS

None

EXAMPLE

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(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_free_dma_handle(9F), wsio_free_shared_mem(9F), wsio_iova_to_phys(9F),
wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F)
NAME
wsio_free_dma_handle - Release a DMA handle.

SYNOPSIS
#include <sys/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 must be called anytime a handle is no longer needed.

RETURN VALUES
None

CONSTRAINTS
None

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 */

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

SEE ALSO
wsio_allocate_dma_handle(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_fastmap_dma_buffer(9F), wsio_free_shared_mem(9F), wsio_flush_shared_mem(9F),
wsio_init_map_context(9F), wsio_iova_to_phys(9F), wsio_map_dma_buffer(9F),
wsio_remap_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F)
wsio_unmap_dma_buffer(9F)
NAME
wsio_free_mem - Free memory allocated by wsio_alloc_mem.

SYNOPSIS
#include <sys/wsio.h>

void wsio_free_mem(
    wsio_mem_handle_t mem_handle,
    wsio_vaddr_t  vaddr
);

PARAMETERS

mem_handle Handle allocated by a call to wsio_alloc_mem_handle.
vaddr Pointer to the allocated buffer.

DESCRIPTION
This WSIO service frees memory allocated by the service wsio_alloc_mem.

RETURN VALUES
None

CONSTRAINTS
May be called in user or interrupt context.
Do not call while holding a spinlock.

EXAMPLES
/* mem_handle is a handle allocated using
   wsio_alloc_mem_handle */
/* vaddr is a pointer to a block of memory allocated using
   wsio_alloc_mem */

wsio_free_mem(mem_handle, vaddr);

SEE ALSO
wsio_alloc_mem(9F), wsio_alloc_mem_handle(9F), wsio_free_mem_handle(9F)
**NAME**

wsio_free_mem_handle - Destroy handle allocated by wsio_alloc_mem_handle.

**SYNOPSIS**

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

void wsio_free_mem_handle(
    wsio_mem_handle_t mem_handle
);
```

**PARAMETERS**

*mem_handle* Handle allocated by a call to wsio_alloc_mem_handle.

**DESCRIPTION**

This WSIO service destroys a *mem_handle* that was allocated by a previous call to wsio_alloc_mem_handle.

**RETURN VALUES**

None

**CONSTRAINTS**

May be called in user or interrupt context.

**EXAMPLES**

```c
/* mem_handle is a handle allocated using
   wsio_alloc_mem_handle() */

wsio_free_mem_handle(mem_handle);
```

**SEE ALSO**

wsio_alloc_mem(9F), wsio_alloc_mem_handle(9F), wsio_free_mem(9F).
**NAME**

wsio_free_shared_mem - Release an I/O virtually contiguous DMA buffer.

**SYNOPSIS**

```c
#include <sys/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 wsioAllocate_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 `wsioAllocate_shared_mem`. All parameters to the macro must be the same as those passed to the call that allocated the memory. The `wsio_free_shared_mem` function can be called in a non-blocking context.

**RETURN VALUES**

None

**CONSTRAINTS**

None

**EXAMPLE**

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

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

if (wsioAllocate_shared_mem(isc_entry, dma_handle, buf_size,
    &io_virtual_addr,
    &host_virtual_addr, 0)
    != WSIO_MAP_OK) {
    /* Unable to allocate shared memory, 
```

309
* 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(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_free_dma_handle(9F), wsio_flush_shared_mem(9F), wsio_iova_to_phys(9F),
wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F)
**wsio_get_active_processor_count(9F)**

**NAME**

wsio_get_active_processor_count - Get the number of active CPUs in the system.

**SYNOPSIS**

```c
#include <sys/wsio.h>
int wsio_get_active_processor_count(void);
```

**PARAMETERS**

None

**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**

None

**EXAMPLE**

```c
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(9F)`
NAME

wsio_get_all_registers - Get an array of all available device registers.

SYNOPSIS

#include <sys/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

None

EXAMPLE

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(9F), wsio_read_regXX(9F), wsio_unmap_reg(9F), wsio_write_regXX(9F)
wsio_get_attribute(9F)

NAME

wsio_get_attribute -- Gets an attribute registered with an interface.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_get_attribute(
    IN struct isc_table_type *isc,
    IN char *name,
    OUT uintptr_t *value,
    IN wsio_attrib_flags_t flags
);

PARAMETERS

isc The isc handle of the interface the attribute is associated with.
flags Flags indicating behavior of attribute.
name A character string representing the name of the property.
value The data is returned in the buffer referenced by value.
size The number of bytes in the data is returned.

DESCRIPTION

This service is used to retrieve the current value of an attribute associated with the interface identified by the isc parameter. The value returned depends upon how the attribute was created. If the attribute was created with the WSIO_COPYDATA flag, the caller should pass the same flag into wsio_get_attribute, and provide a buffer large enough to copy the data into. If the WSIO_COPYDATA flag was not set, the immediate data is returned. The size parameter indicates how many bytes were transferred.

RETURN VALUES

WSIO_OK The attribute data is returned.
WSIO_ERROR An error occurred.

CONSTRAINTS

None

SEE ALSO

wsio_create_attribute(9F), wsio_modify_attribute(9F), wsio_destroy_attribute(9F),
wsio_sizeof_attribute(9F).
NAME

wsio_get_ioports - Obtain the addresses and sizes of I/O ports.

SYNOPSIS

#include <sys/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

None

EXAMPLE

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(9F), wsio_port_inXX(9F), wsio_port_outXX(9F)
NAME

wsio_get_isc - Get the ISC structure pointer for a device file.

SYNOPSIS

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

int wsio_get_isc(
    dev_t dev,
    struct isc_table_type **isc_ptr,
    wsio_drv_info_t *wsio_drv_info
);  

PARAMETERS

dev    The device file of the hardware for which we want an ISC pointer.
isc_ptr A pointer to the location for the routine to put a pointer to the ISC structure.
wsio_drv_info A pointer to the wsio_drv_info_t header structure of the driver, used
to decode dev. If a NULL value is passed in this field, wsio_get_isc will
use the wsio_drv_info_t structure of the character device (not block) whose major number matches that of the dev argument.

DESCRIPTION

The wsio_get_isc WSIO function finds an ISC table entry associated with the nearest interface ancestor of the device specified by dev, assuming that dev uses the HP-UX Converged I/O minor number format.

RETURN VALUES

1 Successful completion. The ISC pointer found is returned in isc_ptr.
0 Failure. The ISC could not be found.

CONSTRAINTS

None
NAME

wsio_get_processor_count - Get the number of CPUs in the system.

SYNOPSIS

#include <sys/wsio.h>

int wsio_get_processor_count(void);

PARAMETERS

None

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 of allocated.

RETURN VALUES

The number of processors.

CONSTRAINTS

None

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(9F)
NAME

wsio_get_relationship - Returns the specified relative in the hierarchical tree.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_get_relationship ( 
    IN struct isc_table_type *isc, 
    IN wsio_relationship_t which_relation, 
    IN struct isc_table_type *prev, 
    OUT struct isc_table_type **next 
);

PARAMETERS

isc The isc handle of the HBA to be queried.
which_relation The relation to get, PARENT or CHILD.
prev The isc handle of the last relation to be returned.
next When called, a pointer to an isc handle. On return, the isc handle of the next member in the relationship.

DESCRIPTION

This service is called to get the parents or children of an interface. The first parameter is the isc handle of the interface to query. The which_relation parameter indicates what relationship to query. Valid values are WSIO_ISC_GET_PARENT or WSIO_ISC_GET_CHILD. The prev parameter can be used to indicate the last relation returned if the isc has multiple parents or children. If prev is NULL, the isc of the first child or parent is returned; else, the isc of the next child or parent is returned. The caller must make no assumptions about the order in which members are returned. The last parameter is where the isc of the next relation is returned.

In the following example the caller gets the isc of each if the children of an HBA.

RETURN VALUES

WSIO_OK The next isc is returned.
WSIO_ERROR An error occurred

CONSTRAINTS

None

EXAMPLE

struct isc_table_type *prev, *next;
struct isc_table_type hba_isc;

wsio_ret_code_t ret;
prev = next = NULL;
hba_isc = my_hba_isc;
do { 
    /* Get the interface child */
    ret = wsio_get_relationship ( hba_isc, WSIO_ISC_GET_CHILD, prev, &next);
    ...
    /* Set the prev to next, if intended to get another child */
    prev = next;
} while (ret == WSIO_OK);

SEE ALSO
**wsio_get_system_params**

**NAME**

wsio_get_system_params - Get information about the system.

**SYNOPSIS**

#include <sys/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  
The 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 I/O 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**

None

**EXAMPLE**

```c
int ret;
wsio_unintptr_t value;
ret = wsio_get_system_params(isc_entry,
    WSIO_CACHELINE_SIZE, &value);

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

wsio_hwpath_to_isc - Returns the HBA at the specified hardware path.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret__code_t wsio_hwpath_to_isc(
    IN struct isc_table_type *ancestor,
    IN char *path,
    IN/OUT struct isc_table_type **isc
);

PARAMETERS

ancestor Either NULL or the isc handle of an ancestor to start the hw_path from.
path A string with the hardware path.
isc IN: A pointer to an uninitialized isc handle.
OUT: The handle initialized with the isc of the target.

DESCRIPTION

This service is used to find the isc structure of an interface at the path specified by the isc and path parameters. If the ancestor parameter is not NULL, path is assumed to be relative to the hardware path associated with the isc; otherwise, path is treated as an absolute path. If successful, the isc handle of the interface is returned. An interface can be of type WSIO_INTERFACE, WSIO_VIRT_BUS, and WSIO_TRANS.

RETURN VALUES

WSIO_OK The isc handle is returned.
WSIO_ERROR An error occurred.

CONSTRAINTS

None

SEE ALSO

wsio_isc_to_hwpath(9F)
NAME

wsio_init_map_context - Initialize the context used for DMA mapping.

SYNOPSIS

```c
#include <sys/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 must be used for a set of DMA mappings that are all going to be unmapped at the same time. This context must be initialized before its first use. If the mapping that is being performed is independent of all other mappings, a context is not necessary and NULL must be passed to the mapping function in the context field.

RETURN VALUES

None

CONSTRAINTS

None

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(9F), wsio_dma_pass_thru(9F), wsio_fastmap_dma_buffer(9F),
wsio_free_dma_handle(9F), wsio_iova_to_phys(9F), wsio_map_dma_buffer(9F),
wsio_remap_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F),
wsio_unmap_dma_buffer(9F)
wsio_install_driver(9F)

NAME

wsio_install_driver – Install a driver’s header structure into the WSIO CDIO.

SYNOPSIS

#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

None

EXAMPLES

/* 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 */}
/* Declare the WSIO driver-specific fields */
static wsio_drv_data_t beep_data = {
    "hil",    /* char *drv_path   match probes-drivers */
    T_DEVICE, /* sbit8 drv_type   type of H/W: dev or IF */
    DRV_CONVERGED,
    /* ubit32 drv_flags DRV_CONV... or NOT_C... */
    NULL,
    /* int (*drv_minor_build)() minor# formatter */
    NULL
    /* int (*drv_minor_decode)() interpreter */
};

static wsio_drv_info_t beep_wsio_info = {
    &beep_info, /* drv_info_t *drv_info */
    &beep_ops, /* drv_ops_t *drv_ops driver entry points */
    &beep_data /* wsio_drv_data_t *drv_data */
};

beep_install()
{
    /* register driver with WSIO and return any error */
    return( wsio_install_driver( beep_wsio_info ) );
}

SEE ALSO
install_driver(9F)
NAME

wsio_install_drv_event_handler - Install a driver's event handler.

SYNOPSIS

#include <sys/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 must 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

None

EXAMPLE

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;
    }
    /* 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;
    }
SEE ALSO

wsio_query_supported_function(9E), wsio_reg_drvcapability_mask(9F),
wsio_uninstall_drv_event_handler(9F)
 wsio_intr_activate(9F)

NAME

wsio_intr_activate - Enable an interrupt object.

SYNOPSIS

#include <sys/wsio.h>

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

PARAMETERS

isc Pointer to the driver's isc_table entry.

obj Interrupt object to enable.

DESCRIPTION

The wsio_intr_activate WSIO function activates an interrupt object that was allocated with
wsio_intr_alloc. The interrupt object must be activated before the system will call the device
driver's ISR (as specified in wsio_intr_alloc). It is assumed that (if possible) the device will
not generate interrupts until after this function is called. The wsio_intr_deactivate or
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
WSIO_INTR_ACTIVATED, without modifying the interrupt object.

RETURN VALUES

WSIO_OK Operation succeeded.

WSIO_ERROR Failure. Could not enable obj.

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

None

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);
}

SEE ALSO

wsio_intr_alloc(9F), wsio_intr_deactivate(9F), wsio_intr_deactivate_nowait(9F),
wsio_intr_set_cpu_spec(9F), wsio_intr_set_line(9F)
NAME

wsio_intr_alloc - Allocate an interrupt object.

SYNOPSIS

#include <sys/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 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; use it with care.

The flags parameter must 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.

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

None

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(9F)
wsio_intr_deactivate(9F)

NAME

wsio_intr_deactivate - Disable an interrupt object.

SYNOPSIS

#include <sys/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.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_INTR_DEACTIVATED obj not active.
WSIO_PARM_ERROR Invalid parameters.
WSIO_ERROR Could not deactivate obj.

CONSTRAINTS

Do not call 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);
}

SEE ALSO
wsio_intr_activate(9F), wsio_intr_alloc(9F), wsio_intr_deactivate_nowait(9F), wsio_intr_set_cpu_spec(9F),
wsio_intr_set_line(9F),
NAME

wsio_intr_deactivate_nowait - Disable an interrupt object with callback.

SYNOPSIS

#include <sys/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 call back function passed in.
WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

None

EXAMPLE

/* 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(9F), wsio_intr_alloc(9F), wsio_intr_deactivate(9F), wsio_intr_set_cpu_spec(9F),
wsio_intr_set_line(9F)
wsio_intr_free(9F)

NAME

wsio_intr_free - Free an interrupt object.

SYNOPSIS

#include <sys/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 VALUE

WSIO_OK Successful completion.

WSIO_INTR_ACTIVATED Must call wsio_intr_deactivate or wsio_intr_deactivate_nowait first.

WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

Do not call 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(9F)
NAME

wsio_intr_get_assigned_cpu - Get the currently assigned CPU for the interrupt object.

SYNOPSIS

#include <sys/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.

The INTR_ATTR_ASSIGNED_CPU attribute is not currently defined in the BN-CDIO specification, 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

None

EXAMPLE

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
int status;
intptr_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(9F), wsio_intr_alloc(9F), wsio_intr_set_cpu_spec(9F)
NAME

wsio_intr_get_attribute - Get the attribute value for an interrupt object.

SYNOPSIS

#include <sys/wsio.h>

int wsio_intr_get_attribute(
    IN struct isc_table_type *isc,
    IN wsio_intr_object_t *iobj,
    IN wsio_intr_attrib_t attrib,
    OUT intptr_t attr_val
);

PARAMETERS

isc Pointer to the ISC table entry for the driver instance.
attrib Type of interrupt attribute to query.
iobj WSIO interrupt object.
attr_val Current value of the attribute.

DESCRIPTION

This service returns the current value of an interrupt attribute for the specified interrupt object. Valid interrupt attributes that can be queried include:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value Returned/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_INTR_ATTR_ENABLED</td>
<td>WSIO_INTR_ATTR_VAL_SET, interrupt is enabled;</td>
</tr>
<tr>
<td></td>
<td>WSIO_INTR_ATTR_VAL_CLEAR, interrupt is disabled.</td>
</tr>
<tr>
<td>WSIO_INTR_ATTR_ORDERED</td>
<td>WSIO_INTR_ATTR_VAL_SET, interrupts are ordered with</td>
</tr>
<tr>
<td></td>
<td>respect to DMA; WSIO_INTR_ATTR_VAL_CLEAR,</td>
</tr>
<tr>
<td></td>
<td>interrupts are not ordered with respect to DMA.</td>
</tr>
<tr>
<td>WSIO_INTR_ATTR_LEVEL</td>
<td>WSIO_INTR_EDGE_SENSITIVE, interrupt is edge</td>
</tr>
<tr>
<td></td>
<td>triggered; WSIO_INTR_LEVEL_SENSITIVE, interrupt is</td>
</tr>
<tr>
<td></td>
<td>level triggered.</td>
</tr>
<tr>
<td>WSIO_INTR_ATTR_SHARE_VEC</td>
<td>WSIO_INTR_SHARED, driver's ISR is shared;</td>
</tr>
<tr>
<td></td>
<td>WSIO_INTR_EXCLUSIVE, driver's ISR is not shared.</td>
</tr>
</tbody>
</table>

This interface can be called from a non-blocking context.

RETURN VALUES

WSIO_OK Attribute was successfully changed.
WSIO_INVALID_ISC Not a valid ISC structure.
WSIO_INTR_INV_OBJ Not a valid interrupt object.
WSIO_NOT_IMPLEMENTED Specified attribute can not be queried.
WSIO_PARM_ERROR The attrib parameter is not valid.

CONSTRAINTS

None

EXAMPLES

wsio_intr_object_t iobj;
wsio_intr_attrib_t attrib;
intptr_t attr_val;

attrib = WSIO_INTR_ATTR_LEVEL;
/* Find out if the interrupt associated with iobj is edge
 * or level triggered. 
 */
status = wsio_intr_get_attribute (isc, iobj, attrib,
                                   &attr_val);

SEE ALSO
wsio_intr_alloc(9F), wsio_intr_free(9F), wsio_intr_activate(9F), wsio_intr_deactivate(9F),
wsio_intr_deactivate_nowait(9F), wsio_intr_set_cpu_spec(9F), wsio_intr_set_irq_line(9F),
wsio_intr_set_attribute(9F)
wsio_intr_get_irq_line(9F)

NAME

wsio_intr_get_irq_line - Get the interrupt line number.

SYNOPSIS

#include <sys/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 is to 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.
WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

None

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(9F), wsio_intr_alloc(9F), wsio_intr_set_line(9F)
NAME

wsio_intr_get_txn_info - Get the transaction address and data value.

SYNOPSIS

#include <sys/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

None

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(9F), wsio_intr_alloc(9F), wsio_intr_set_cpu_spec(9F)
NAME

wsio_intr_set_attribute - Set the attribute value for an interrupt object.

SYNOPSIS

#include <sys/wsio.h>

int wsio_intr_set_attribute(
  IN struct isc_table_type *isc,
  IN wsio_intr_object_t *iobj,
  IN wsio_intr_attrib_t attrib,
  OUT intptr_t attr_val
);

PARAMETERS

isc          Pointer to the device ISC structure.
iobj         Interrupt object returned by the wsio_intr_alloc call.
attrib       The interrupt attribute to change.
attr_val      Value to be assigned to the attribute.

DESCRIPTION

This service is called to set an attribute value associated with the interrupt object. The interrupt
 corresponding to iobj must be disabled before making this call. This can be accomplished by
calling the wsio_intr_deactivate or wsio_intr_deactivate_nowait call. Refer to the
respective manpages for further information.

This routine can be called from a non-blocking context.
Currently there are no attributes that can be set using this call.

RETURN VALUES

WSIO_OK       The attribute was successfully changed.
WSIO_INVALID_ISC Not a valid ISC structure.
WSIO_INTR_INV_OBJ Not a valid interrupt object.
WSIO_INTR_ACTIVATED The interrupt object is activated, so an attribute can not be sent.
WSIO_NOT_IMPLEMENTED The specified interrupt attribute is not implemented.
WSIO_PARM_ERROR The one or more arguments passed to the
                  wsio_intr_set_attribute call were invalid.
WSIO_ERROR    Setting the attribute failed.

CONSTRAINTS

This service must be called on an interrupt object that has been disabled. See the
wsio_intr_deactivate(9F) or wsio_intr_deactivate_nowait(9F) for more information.

EXAMPLES

wsio_intr_object_t iobj;
wsi0_intr_attrib_t attrib;
intptr_t attr_val;

status = wsio_intr_set_attribute ( isc, iobj, attrib,
                      attr_val);
SEE ALSO

wsio_intr_alloc(9F), wsio_intr_free(9F), wsio_intr_activate(9F), wsio_intr_deactivate(9F),
wsio_intr_deactivate_nowait(9F), wsio_intr_set_cpu_spec(9F), wsio_intr_set_irq_line(9F),
wsio_intr_get_attribute(9F)
NAME

wsio_intr_set_cpu_spec - Initialize and distribute transaction based interrupts.

SYNOPSIS

#include <sys/wsio.h>

(
    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 neither of the following flags but
is a txn_addr, the interrupt is bound to this CPU.

WSIO_INTR_CPU_ANY The services select any processor. The device
driver does not care which processor it is
bound to.

WSIO_INTR_CPU_ANY_UNIQUE The services do not select the same processor
for an obj (only one obj per processor for this
isc).

txn_addr Bind to this CPU address.

DESCRIPTION

The wsio_intr_set_cpu_spec WSIO function is used to initialize and distribute transaction
based interrupts.

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

Do not call from the driver_attach routine.
Can be called from the driver_if_init routine.

EXAMPLE

/* Allocate a transaction based interrupt */
wsio_intr_object_t obj;
```c
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(9F), wsio_intr_alloc(9F), wsio_intr_set_cpu_spec(9F)
```
NAME

wsio_intr_set_irq_line - Set the interrupt line number.

SYNOPSIS

#include <sys/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_IRQ_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 must be zero. If the device
generates edge-sensitive interrupts, the flags parameter must be WSIO_INTR_EDGE_SENSITIVE.

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 wsio_intr_activate first.
WSIO_ERROR Failed to set line number.
WSIO_INTR_INV_FLAG Must be zero (level) or WSIO_INTR_EDGE_SENSITIVE.
WSIO_PARM_ERROR Invalid parameters.

CONSTRAINTS

Do not call from the driver_attach routine.
Can be called from the driver_if_init routine.

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

wsio_intr_activate(9F), wsio_intr_alloc(9F), wsio_intr_set_line(9F)
NAME

wsio_io_sync - Perform a sync of shared memory if necessary.

SYNOPSIS

#include <sys/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. The wsio_io_sync function can be called in a non-blocking context.

RETURN VALUES

None

CONSTRAINTS

None

EXAMPLE

wsio_io_sync(isc_entry);

SEE ALSO

WSIO_ORDERED_INTERRUPTS(9F)
NAME

wsio_iova_to_phys - Translate an I/O virtual address to a physical address.

SYNOPSIS

#include <sys/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.
The wsio_iova_to_phys function 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

None

EXAMPLE

caddr_t phys_address = NULL;
phys_address = wsio_iova_to_phys(isc_entry,dma_handle,i0_address);

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

SEE ALSO

wsio_allocate_dma_handle(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_fastmap_dma_buffer(9F), wsio_free_dma_handle(9F), wsio_free_shared_mem(9F),
wsio_flush_shared_mem(9F), wsio_init_map_context(9F), wsio_map_dma_buffer(9F),
wsio_remap_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_unmap_dma_buffer(9F)
NAME

wsio_isc_to_hwpath - Returns the hardware path of the HBA.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_isc_to_hwpath (
    IN struct isc_table_type *isc,
    OUT char *path
);

PARAMETERS

isc IN: The isc handle of the interface.

path IN: A pointer to a string.
    OUT: The string initialized with the hardware path.

DESCRIPTION

This service returns the hardware path of an interface described by the isc parameter. The
hardware path is returned in the path parameter.

RETURN VALUES

WSIO_OK Path returned successfully.
WSIO_ERROR An error occurred.

CONSTRAINTS

None

SEE ALSO

wsio_hwpath_to_isc(9F)
wsio_isc_to_instance(9F)

NAME

wsio_isc_to_instance - Retrieve an instance number of an iotree node.

SYNOPSIS

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

int wsio_isc_to_instance(
    struct isc_table_type *isc,
    hw_path_t *dev_hw_path
);

PARAMETERS

isc      A pointer to the ISC structure associated with the interface card for the device.
dev_hw_path      A pointer to a structure containing device hardware path information relative
to the interface card, or NULL if the card instance is desired.

DESCRIPTION

The wsio_isc_to_instance WSIO function provides an instance number of an iotree node
that is a descendant of the card or device node specified by the isc and dev_hw_path parameters,
according to the following rules:

• If isc is valid and dev_hw_path is NULL, the instance number of an iotree node corresponding
to the interface card is returned.

• If isc is valid and dev_hw_path contains a path to a valid device (relative to the device's
  interface card), the instance number of the iotree node corresponding to the device is returned.
  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 must return WSIO_ERROR.

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

None
WSIO_LITTLE_ENDIAN(9F)

NAME

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

SYNOPSIS

#include <sys/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 must 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

None

EXAMPLE

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

SEE ALSO

WSIO_BIG_ENDIAN(9F)
NAME

wsio_map_cfg_handle - Obtain a configuration space access handle.

SYNOPSIS

#include <sys/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

Do not call in an interrupt context.

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);
}

/* The configuration space handle is now in the handle variable */

SEE ALSO

wsio_cfg_inXX(9F), wsio_cfg_outXX(9F), wsio_unmap_cfg_handle(9F)
NAME

wsio_map_dma_buffer - Map an existing memory object for packet DMA.

SYNOPSIS

#include <sys/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 wsio_allocate_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>
<tr>
<td>WSIO_DMA_LOCK</td>
<td>In some implementations this allows atomic access to memory for devices using bus-lock primitives.</td>
</tr>
<tr>
<td>WSIO_DMA_FLUSH_ON_USE</td>
<td>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.</td>
</tr>
<tr>
<td>WSIO_DMA_IGN_ALIGNMENT</td>
<td>Indicates wsio_map_dma_buffer must not automatically set the IO_SAFE hint for partial cacheline buffers.</td>
</tr>
<tr>
<td>WSIO_DMA_CONTIGUOUS</td>
<td>Indicates wsio_map_dma_buffer must allocate a single contiguous IOVA. If wsio_map_dma_buffer is unable to do this, it returns WSIO_MAP_E_PARAMETER_ERROR. This hint implies WSIO_IO_IGN_ALIGNMENT.</td>
</tr>
<tr>
<td>WSIO_DMA_NO_SEQ</td>
<td>Inhibits hardware prefetching for outbound DMA in some implementations.</td>
</tr>
<tr>
<td>WSIO_DMA_INBOUND</td>
<td>Indicates the buffer is used exclusively for inbound DMA.</td>
</tr>
</tbody>
</table>
WSIO_DMA_OUTBOUND Indicates the buffer is 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 is 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 contains information about the remaining space to be mapped when the call completes.

io_range Pointer to an address or length structure containing 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 must be used. Continuous DMA must 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 must 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 must be initialized via a call to wsio_init_map_context. The same context must 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.

CONSTRAINTS
None

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);
dsio_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(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_fastmap_dma_buffer(9F), wsio_free_dma_handle(9F), wsio_free_shared_mem(9F),
wsio_flush_shared_mem(9F), wsio_iova_to_phys(9F), wsio_init_map_context(9F),
wsio_remap_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F),
wsio_unmap_dma_buffer(9F)
wsio_map_port(9F)

NAME

wsio_map_port - Obtain an I/O port handle.

SYNOPSIS

#include <sys/wsio.h>

int wsio_map_port(
    struct isc_table_type *isc,
    int32_t port_addr,
    sizet_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_ioports.
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. Do not call the wsio_map_port function in a non-blocking context.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR There was a parameter error.

CONSTRAINTS

Do not call in an interrupt context.

EXAMPLE

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);
}

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

SEE ALSO

wsio_get_ioports(9F), wsio_port_inXX(9F), wsio_port_outXX(9F), wsio_unmap_port(9F)
NAME

wsio_map_reg - Map device registers to host memory space.

SYNOPSIS

#include <sys/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 must 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. Do not call the wsio_map_reg function in a non-blocking context.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR Could not create the mapping.

CONSTRAINTS

Do not call 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(9F), wsio_read_regXX(9F), wsio_write_regXX(9F), wsio_unmap_reg(9F)
NAME
wsio_mem_alloc_attrib_t -- Memory allocation attributes.

SYNOPSIS
#include <sys/wsio.h>

PARAMETERS
WSIO_OPTIMIZE_FOR_DEVICE  Allocate memory close to the device.
WSIO_OPTIMIZE_FOR_CPU     Allocate memory close to the current CPU. This is the default behavior.
WSIO_32BIT_MEMORY         The buffer must be allocated before 4 GB.
WSIO_IO_CONTIGUOUS        On platforms without an I/O PDIR, physically contiguous memory is allocated.
WSIO_ALIGN_ON_SIZE        With this attribute, buffers are aligned with the same alignment as their size.

DESCRIPTION
Flags passed to the wsio_alloc_mem_handle service to indicate what type of memory will be allocated with the handle.

RETURN VALUES
WSIO_ALLOC_OK             Indicates that the buffer was allocated.
WSIO_ALLOC_OUT_OF_RESOURCES Unable to allocate the specified resources.
WSIOINVAL_PARAM           A parameter was not valid.

CONSTRAINTS
None

SEE ALSO
NAME

wsio_modify_attribute - Modifies an existing attribute.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_modify_attribute(
    IN struct isc_table_type *isc,
    IN char *name,
    IN uintptr_t *value,
    IN size_t size,
    wsio_attrib_flags_t flags
);

PARAMETERS

isc The isc handle of the interface that the attribute will be is associated with.
flags Flags indicating behavior of attribute.
name A character string representing the name of the attribute.
value A pointer to the attributes current data.
size The size of the data in bytes.

DESCRIPTION

This service is called to modify the value of an attribute associated with an interface. The isc handle of the interface is passed in as the first parameter. The new data for the attribute is defined by the value and size parameters. The size parameter indicates the size of the new data. If size is greater than the original, the service might fail or block if WSIO_WAIT_OK is specified in the flags.

The valid attribute flags are listed:

<table>
<thead>
<tr>
<th>wsio_attrib_flags_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_COPYDATA</td>
<td>If set, value is assumed to contain an address that references a data buffer and the contents of the buffer is copied; otherwise, value is assumed to contain the immediate data that is saved.</td>
</tr>
<tr>
<td>WSIO_WAIT_OK</td>
<td>If resources are not available the call will block until they are.</td>
</tr>
<tr>
<td>WSIO_ATTR_EXPORT</td>
<td>The attribute will be exported to any children.</td>
</tr>
</tbody>
</table>

RETURN VALUES

WSIO_OK Path returned successfully.
WSIO_ERROR An error occurred.

CONSTRAINTS

Do not call on ICS.

SEE ALSO

wsio_create_attribute(9F), wsio_get_attribute(9F), wsio_destroy_attribute(9F), wsio_sizeof_attribute(9F).
NAME

wsio_msi_alloc - Allocate a Message-Signaled Interrupt object.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_msi_alloc(
    struct isc_table_type *isc,
    wsio_msi_type_t type,
    int vector_cnt,
    wsio_msi_hints_t hints,
    wsio_msi_hndl_t *hndl
);

PARAMETERS

isc Pointer to the driver's isc_table entry.

type Type of interrupt object to allocate, either WSIO_MSI_TYPE or
      WSIO_MSI_X_TYPE.

vector_cnt Number of interrupt vectors requested.

hints Reserved for future use.

hndl Returned interrupt object.

DESCRIPTION

The wsio_msi_alloc service is called to allocate an MSI/MSI-X interrupt object.

Valid values for type are shown in the following table:

<table>
<thead>
<tr>
<th>wsio_msi_type_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MSI_TYPE</td>
<td>Allocate an interrupt object for MSI type interrupts.</td>
</tr>
<tr>
<td>WSIO_MSI_X_TYPE</td>
<td>Allocate an interrupt object for MSI-X type interrupts.</td>
</tr>
</tbody>
</table>

The value passed in for vector_cnt must not exceed the number of vectors supported by the
MSI/MSI-X function. The latter information can be obtained using the wsio_msi_capability
service. Drivers must always call wsio_msi_capability to determine the hardware and
platform capabilities or limitations before calling this service. For an object of WSIO_MSI_TYPE,
vector_cnt must be equal to one. The service will return WSIO_ERROR if it is not.

The hints field is reserved for future hints. It must be set to zero.

If the call is successful the handle for the new MSI object is returned in the hndl parameter.

A PCI card cannot enable both MSI and MSI-X functions at the same time. If wsio_msi_alloc
is called to allocate an interrupt object of one type, a subsequent call to wsio_msi_alloc to
allocate an interrupt object of the other type will result in an error.

Any driver using MSI or MSI-X interrupts must register for the WSIO_EVENT_OFFLINE_CPU
and WSIO_EVENT_MSI_INTR_MIGR events. If the isc has not registered for both of those events,
wsio_msi_alloc returns WSIO_ERROR.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR  A MSI/MSI-X interrupt object has already been allocated for this PCI function, or an error occurred.

WSIO_INVALID_ISC  The isc is not valid.

WSIO_INVALID_PARM  The type or hints parameter is invalid, the PCI function does not support the given type, or vector_cnt is an invalid value.

WSIO_NOT_SUPPORTED  The platform does not have support for MSI/MSI-X features.

CONSTRAINTS
Do not call while holding a spinlock or while executing in interrupt context.

EXAMPLE

```c
wsio_ret_code_t ret;
int vector_cnt;
wsio_msi_hndl_t msi_hndl;

/* Allocate a maximum of 5 interrupt vectors */
ret = wsio_msi_capability(isc, WSIO_MSI_X_CAPABILITY, &vector_cnt);
if (vector_cnt > 0) {
    if (vector_cnt > 5) vector_cnt = 5;
    ret = wsio_msi_alloc(isc, WSIO_MSI_X_TYPE, vector_cnt, 0, &msi_hndl);
}
```

SEE ALSO

wsio_msi_assign(9F), wsio_msi_capability(9F), wsio_msi_disable(9F), wsio_msi_enable(9F),
wsio_msi_free(9F), wsio_msi_get_cpus(9F), wsio_msi_query(9F), wsio_msi_resize(9F)
NAME

wsio_msi_assign - Initialize Message-Signaled Interrupt vectors.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_msi_assign(
    wsio_msi_hndl_t hndl,
    wsio_msi_assign_type_t assign_type,
    wsio_msi_setinfo_t *vec_info,
    int qualifier,
    size_t num
);

PARAMETERS

hndl Handle of an MSI/MSI-X interrupt object.
assign_type How to set the per-vector information.
vec_info Array of wsio_msi_setinfo_t structures, describing the vector values.
qualifier The starting vector, or a CPU identifier. Its meaning depends on the assign_type.
num The number of vectors to modify, or the number of elements in vec_info. Its meaning depends on the assign_type.

DESCRIPTION

The wsio_msi_assign service initializes a set of interrupt vectors belonging to an MSI/MSI-X interrupt object. All vectors being modified must be disabled, otherwise wsio_msi_assign will return an error.

Values for assign_type are shown in the table below.

Table 2-7 Values for assign_type

<table>
<thead>
<tr>
<th>wsio_msi_assign_type_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MSI.Assign.Range</td>
<td>Indicates that a contiguous range of vectors must be modified. The parameters qualifier indicates the start of the range and num indicates the size.</td>
</tr>
<tr>
<td>WSIO_MSI.CPU.ID</td>
<td>Indicates that the parameter qualifier is a CPU identifier. Only those vectors assigned to the specified CPU will be modified.</td>
</tr>
<tr>
<td>WSIO_MSI.Assign.Per.VEC</td>
<td>Indicates that the vec_info array contains an individual entry for each vector to be modified. This flag can be OR'd together with either WSIO_MSI.Assign.Range or WSIO_MSI.CPU.ID.</td>
</tr>
<tr>
<td>WSIO_MSI.Assign.All</td>
<td>Indicates that the vec_info array contains a single entry that must be applied to all the specified vectors. This flag can be OR'd together with either WSIO_MSI.Assign.Range or WSIO_MSI.CPU.ID.</td>
</tr>
</tbody>
</table>

Valid flag combinations for assign_type include:

(WSIO_MSI.Assign.Range | WSIO_MSI.Assign.Per.VEC) Assign a range of contiguous vectors and the vector array has individual assignments for each. There is a 1 to 1 correspondence between the entries in the vec_info array and the number of vectors to modify hence the parameter num indicates the number of entries in the vec_info array and the number of vectors to modify starting from the vector indicated by qualifier.
Assign a range of contiguous vectors. The vector array has a single assignment for all. The `vec_info` array contains a single entry and `num` indicates the number of vectors to modify starting from the vector indicated by `qualifier`.

Assign all vectors that are currently assigned to the CPU specified in `qualifier`. The vector array has individual assignments for each vector hence `num` indicates the total number of entries in the `vec_info` array.

Assign all vectors that are currently assigned to the CPU specified in `qualifier`. The vector array has a single assignment for all. The value of `num` is ignored.

The `vec_info` parameter points to an array of structures of type `wsio_msi_setinfo_t`. Its definition is as follows:

```c
typedef struct wsio_msi_setinfo {
    int           mask;
    int          (isr*)(());
    void         *isr_arg;
    wsio_cpu_id_t cpu_id;
} wsio_msi_setinfo_t;
```

- `mask` The bit-wise OR of flag values that describe what other fields are valid, taken from:
  - `WSIO_MSI_SET_CPU` The `cpu_id` field is valid.
  - `WSIO_MSI_SET_ISR` The `isr` and `isr_arg` fields are valid.
- `isr` Pointer to the driver's interrupt service routine.
- `isr_arg` Driver defined parameter passed as the first parameter to `isr`. Typically, `isc` is passed as `isr_arg`.
- `cpu_id` CPU identifier. This is typically obtained from a call to `wsio_msi_get_cpus`.

For example, the caller could specify that all fields in the structure are valid by setting `mask` to `(WSIO_MSI_SET_CPU | WSIO_MSI_SET_ISR)` and then initializing all fields in the `wsio_msi_setinfo_t` structure.

Alternatively, the caller could specify that only the CPU identifier is to be set by setting `mask` to `WSIO_MSI_CPU_SET_ID`, and then only initializing the `cpu_id` in each `wsio_msi_setinfo_t` structure.

If multiple vectors of an MSI type interrupt object are being assigned then only the `cpu_id` in the first `wsio_msi_setinfo_t` structure will be used. This is because all of the MSI vectors are directly towards the same CPU. Only the data portion of the vector differs.

**RETURN VALUES**

- `WSIO_OK` Successful completion.
- `WSIO_ERROR` One or more vectors are active, or an error occurred.
- `WSIO_INVALID_PARM` Invalid value for a parameter was passed in.
- `WSIO_NOT_SUPPORTED` The platform does not have support for MSI/MSI-X features.

**CONSTRAINTS**

Do not call this service while holding a spinlock.

If CPU assignments are being modified, do not call this service while executing in interrupt context.

**EXAMPLES**

In the following example, the caller initializes 10 contiguous vectors of an MSI-X object starting at vector 5 with the same values.
wsio_msi_setinfo_t vec_vals;

/* Only 1 vector entry, so it is not an array */
vec_vals.mask = WSIO_MSI_SET_CPU | WSIO_MSI_SET_ISR;
vec_vals.isr = my_drv_isr;
vec_vals.isr_arg = my_isr_arg;
vec_vals.cpu_id = cpu_id;

/* Vectors 5 thru 14 will be updated with the specified values */
wsio_msi_assign(hndl, (WSIO_MSI_ASSIGN_RANGE | WSIO_MSI_ASSIGN_ALL),
                   &vec_vals, 5, 10);

In the following example, the caller only wants to change the CPU assignment of all vectors
assigned to old_cpu_id.

wsio_msi_setinfo_t vec_vals;

/* Only the CPU id is changing, so set the mask appropriately */
vec_vals.mask = WSIO_MSI_SET_CPU;
vec_vals.cpu_id = new_cpu_id; /* Disable all vectors assigned to the old CPU id */
wsio_msi_disable(hndl, old_cpu_id, 0, 0); /* Change only the vectors associated with the old CPU */
type = (WSIO_MSI_CPU_ID | WSIO_MSI_ASSIGN_ALL);
wsio_msi_assign(hndl, type, &vec_vals, old_cpu_id, 0); /* Enable all the vectors assigned to the new CPU id */
wsio_msi_enable(hndl, new_cpu_id, 0, 0);

SEE ALSO

wsio_msi_alloc(9F), wsio_msi_capability(9F), wsio_msi_disable(9F), wsio_msi_enable(9F),
wsio_msi_free(9F), wsio_msi_get_cpus(9F), wsio_msi_query(9F), wsio_msi_resize(9F)
**NAME**

wsio_msi_capability - Report the MSI and MSI-X capabilities of a PCI interface card.

**SYNOPSIS**

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

wsio_ret_code_t wsio_msi_capability(
    struct isc_table_type *isc,
    wsio_msi_plat_cap_t capability,
    int *value
);
```

**PARAMETERS**

- **isc** Pointer to the driver's isc_table entry.
- **capability** Type of capability to query.
- **value** Returned value indicating the MSI/MSI-X capability.

**DESCRIPTION**

The `wsio_msi_capability` service queries the MSI/MSI-X capabilities of a PCI card. The capabilities returned are determined by both the PCI card’s MSI/MSI-X hardware and any further limitations imposed by the platform.

Drivers must always call this service to determine the capabilities before calling `wsio_msi_alloc` to allocate an MSI or MSI-X object.

Valid values for `capability` are shown in the following table:

**Table 2-8 Values for capability**

<table>
<thead>
<tr>
<th><code>wsio_msi_plat_cap_t</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MSI_CAPABILITY</td>
<td>A returned <code>value</code> of 0 indicates that the PCI card does not support MSI. A <code>value</code> of 1, 2, 4, 8, 16 or 32 indicates the maximum number of vectors supported by the MSI function of the card.</td>
</tr>
<tr>
<td>WSIO_MSI_X_CAPABILITY</td>
<td>The service returns a number in the range of 0 to 2048 in the parameter <code>value</code>. A value of 0 indicates that MSI-X type interrupts are not supported; otherwise, a number between 1 to 2048, inclusive, indicates the maximum number of entries in the MSI-X vector table for this PCI device.</td>
</tr>
<tr>
<td>WSIO_MSI_MASK_CAPABILITY</td>
<td>The service returns a value indicating if the hardware supports per-vector masking for MSI and MSI-X. Valid results for <code>value</code> are: 0: Masking is not supported. 1: Masking is supported for MSI only. 2: Masking is supported for MSI-X only. 3: Masking is supported for both MSI and MSI-X.</td>
</tr>
</tbody>
</table>

The value returned for `WSIO_MSI_CAPABILITY` indicates if MSI type interrupts are supported and if so the maximum number of contiguous vectors that can be allocated. The platform determines this by first reading the Capability_ID register of the capabilities structure and if MSI is supported the Multiple_Message_Capable field of the Message_Control register of the MSI capabilities structure. The platform may impose further restrictions on the maximum number of vectors supported; in particular, the current MSI implementation on HP-UX only supports 1 vector.

The value returned for `WSIO_MSI_X_CAPABILITY` indicates if MSI-X type interrupts are supported and if so the size of the MSI-X table. The underlying platform code obtains this
information from the Capability_ID register of the capabilities structure and if MSI-X is supported the MSI-X_Table_Size field of the Message_Control register of the MSI capabilities structure. The value returned for WSIO_MSI_MASK_CAPABILITY indicates if per-vector masking is supported by the underlying hardware.

No hardware is initialized at this point.

**RETURN VALUES**

- WSIO_OK: Successful completion.
- WSIO_ERROR: An error occurred.
- WSIO_INVALID_ISC: The isc is not valid.
- WSIO_INVALID_PARM: The capability parameter contains invalid options, or the value parameter is NULL.
- WSIO_NOT_SUPPORTED: The platform does not have support for MSI/MSI-X features.

**CONSTRAINTS**

None

**EXAMPLE**

```c
int vector_cnt;

/* Determine if MSI-X type interrupts are supported */
wsio_msi_capability(my_isc, WSIO_MSI_X_CAPABILITY, &vector_cnt);
if (vector_cnt > 0) {
    printf("MSI-X table size == %d\n", vector_cnt);
}
else {
    /* MSI-X interrupts are not supported; are MSI type interrupts supported? */
    wsio_msi_capability(my_isc, WSIO_MSI_CAPABILITY, &vector_cnt);
    if (vector_cnt > 0) {
        printf("MSI interrupts vector count == %d\n", vector_cnt);
    }
    else {
        printf("Neither MSI nor MSI-X supported by card, using LBI\n");
    }
}
```

**SEE ALSO**

wsio_msi_alloc(9F), wsio_msi_assign(9F), wsio_msi_disable(9F), wsio_msi_enable(9F), wsio_msi_free(9F), wsio_msi_get_cpus(9F), wsio_msi_query(9F), wsio_msi_resize(9F)
NAME

wsio_msi_disable - Disable selected MSI/MSI-X interrupt vectors.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_msi_disable(
    wsio_msi_hndl_t hndl,
    int type,
    int first,
    size_t cnt
);

PARAMETERS

hndl    Handle of an MSI/MSI-X interrupt object.
type    Criterion to select the vectors to be disabled, either WSIO_MSI_VECTOR_RANGE,
        WSIO_MSI_ALL_VECTORS, or a valid CPU identifier.
first   First vector to disable. This is only examined if type is WSIO_MSI_VECTOR_RANGE.
cnt     Number of vectors to disable. This is only examined if type is
        WSIO_MSI_VECTOR_RANGE.

DESCRIPTION

The wsio_msi_disable service is called to disable individual MSI/MSI-X interrupt vectors. The parameter hndl is the MSI-X handle.

The parameter type indicates the criterion for selecting the vectors to disable:

- If type is WSIO_MSI_VECTOR_RANGE, first and cnt indicate a contiguous range of vectors to
disable.
- If type is WSIO_MSI_ALL_VECTORS, all interrupts of the MSI function are disabled. This
does not change any per-vector mask settings, and vectors that were disabled on a per-vector
basis remain disabled when a call to wsio_msi_enable with type set to
WSIO_MSI_ALL_VECTORS is made to re-enable the vectors.
- If type is set to a valid CPU identifier, only those MSI-X vectors that are programmed with
the specified CPU will be disabled. CPU identifiers are distinct from the values of
WSIO_MSI_VECTOR_RANGE and WSIO_MSI_ALL_VECTORS.

The MSI object must support vector masking, otherwise this service returns an error. Drivers can determine if the card supports per-vector masking by calling the service
wsio_msi_capability with the capability parameter set to WSIO_MSI_MASK_CAPABILITY.

When called in process context, wsio_msi_disable waits for ISR completion. When it returns,
the caller is guaranteed that no ISRs are currently executing. However, this guarantee does not apply when this service is called in ICS context or timeout context.

RETURN VALUES

WSIO_OK    Successful completion.
WSIO_ERROR An error occurred.
WSIO_INVALID_PARM Invalid value for type, invalid CPU identifier, or invalid vector range specified by first and cnt parameters.
WSIO_NOT_SUPPORTED The platform does not have support for MSI/MSI-X features.
CONSTRAINTS
Do not call this service while holding a spinlock.

EXAMPLE

wsio_msi_hndl_t msi_hnd1;

/* Disable all vectors on the handle */
wsio_msi_disable(msi_hnd1, WSIO_MSI_ALL_VECTORS, 0, 0);

SEE ALSO

wsio_msi_alloc(9F), wsio_msi_assign(9F), wsio_msi_capability(9F), wsio_msi_enable(9F),
wsio_msi_free(9F), wsio_msi_get_cpus(9F), wsio_msi_query(9F), wsio_msi_resize(9F)
**NAME**

wsio_msi_enable - Enable selected MSI/MSI-X interrupt vectors.

**SYNOPSIS**

```c
#include <sys/wsio.h>
wsio_ret_code_t wsio_msi_enable(
    wsio_msi_hndl_t hndl,
    int type,
    int first,
    size_t cnt
);
```

**PARAMETERS**

- **hndl** Handle of an MSI/MSI-X interrupt object.
- **type** Criterion to select the vectors to be enabled, either `WSIO_MSI_VECTOR_RANGE`, `WSIO_MSI_ALL_VECTORS`, or a valid CPU identifier.
- **first** First vector to enable. This is only examined if **type** is `WSIO_MSI_VECTOR_RANGE`.
- **cnt** Number of vectors to enable. This is only examined if **type** is `WSIO_MSI_VECTOR_RANGE`.

**DESCRIPTION**

The `wsio_msi_enable` service is called to enable MSI/MSI-X interrupt vectors. The parameter **hndl** is the MSI-X handle.

The parameter **type** indicates the criterion for selecting the vectors to enable:

- If **type** is `WSIO_MSI_VECTOR_RANGE`, **first** and **cnt** indicate a contiguous range of vectors to enable. This option is useful during PCI card initialization or when resizing the vector table.
- If **type** is `WSIO_MSI_ALL_VECTORS`, all interrupts of the MSI function and all vectors that are not masked off on a per-vector basis will be re-enabled.
- If **type** is set to a valid CPU identifier, only those MSI-X vectors that are programmed with the specified CPU will be enabled. This option is useful during interrupt migration or when new CPUs come online. Note that CPU identifiers are distinct from the values of `WSIO_MSI_VECTOR_RANGE` and `WSIO_MSI_ALL_VECTORS`.

For MSI type interrupts, if per-vector masking is not supported then this service will simply enable MSI type interrupts on the card. Also for MSI type interrupts, specifying a CPU identifier is not useful, as only a single CPU is assigned as the interrupt address.

**RETURN VALUES**

- **WSIO_OK** Successful completion.
- **WSIO_ERROR** An error occurred.
- **WSIO_INVALID_ISC** The *isc* is not valid.
- **WSIO_INVALID_PARM** Invalid value for **type**, invalid CPU identifier, or invalid vector range specified by **first** and **cnt** parameters.
- **WSIO_NOT_SUPPORTED** The platform does not have support for MSI/MSI-X features.

**CONSTRAINTS**

Do not call this service while holding a spinlock.
There are restrictions on calling `wsio_msi_enable` in interrupt context. It must not be called in interrupt context if the vectors are being enabled for the first time after allocation or for the first time after the ISR or ISR argument have been modified. It may be called in interrupt context if the vectors have already been activated—that is, if the interrupt vector was previously enabled, then masked via `wsio_msi_disable`.

**EXAMPLE**

```c
wsio_msi_hdl_t msi_hdl;

/* Enable all the vectors on the handle */
wsio_msi_enable(msi_hdl, WSIO_MSI_ALL_VECTORS, 0, 0);
```

**SEE ALSO**

`wsio_msi_alloc(9F)`, `wsio_msi_assign(9F)`, `wsio_msi_capability(9F)`, `wsio_msi_disable(9F)`, `wsio_msi_free(9F)`, `wsio_msi_get_cpus(9F)`, `wsio_msi_query(9F)`, `wsio_msi_resize(9F)`
NAME
wsio_msi_free - Free a Message-Signaled Interrupt object.

SYNOPSIS
#include <sys/wsio.h>
wsio_ret_code_t wsio_msi_free(
    wsio_msi_hndl_t *hndl
);

PARAMETERS
hndl  Previously allocated MSI/MSI-X interrupt object.

DESCRIPTION
The wsio_msi_free service deallocates an MSI-X interrupt object. All platform resources
associated with the object are freed. The MSI hardware is disabled (by clearing the Enable bit in
the MessageControl register of the MSI capabilities structure) thus assuring that no MSI or MSI-X
interrupts can be generated.

When called in process context, wsio_msi_free waits for ISR completion. When it returns, the
caller is guaranteed that no ISRs are currently executing. However, this guarantee does not apply
when this service is called in ICS context or timeout context.

RETURN VALUES
WSIO_OK          Successful completion.
WSIO_ERROR       The hndl is not valid, or an error occurred.
WSIO_NOT_SUPPORTED The platform does not have support for MSI/MSI-X features.

CONSTRAINTS
Do not call this service while holding a spinlock or while executing in interrupt context.

EXAMPLE
wsio_ret_code_t ret;
wsio_msi_hndl_t msi_hndl;

/* Free the MSI interrupt object */
ret = wsio_msi_free(msi_hndl);

SEE ALSO
wsio_msi_alloc(9F), wsio_msi_assign(9F), wsio_msi_capability(9F), wsio_msi_disable(9F),
wsio_msi_enable(9F), wsio_msi_get_cpus(9F), wsio_msi_query(9F), wsio_msi_resize(9F)
wsio_msi_get_cpus(9F)

NAME

wsio_msi_get_cpus - Get a list of CPUs available for I/O interrupts.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_msi_get_cpus(
    struct isc_table_type *isc,
    wsio_cpu_id_t *cpu_array,
    int *cnt,
    wsio_msi_cpu_hints_t hints
);

PARAMETERS

isc Pointer to the driver's isc_table entry.

cpu_array Array of CPU identifiers to be filled in by wsio_msi_get_cpus, or NULL.

cnt Number of elements in the cpu_array.

hints Selection criteria for CPUs.

DESCRIPTION

The wsio_msi_get_cpus service is called to get a list of CPUs that are available for I/O interrupts. The caller passes in an array of CPU identifiers in cpu_array and wsio_msi_get_cpus initializes the array with valid CPU identifiers. The cnt parameter indicates the size of the array when passed in and the number of valid entries on return.

Valid values for hints are shown in the following table:

<table>
<thead>
<tr>
<th>wsio_msi_cpu_hints_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MSI_CPU_LOCAL</td>
<td>Return only CPUs local to the HBA, as described by the isc parameter.</td>
</tr>
<tr>
<td>WSIO_MSI_GET_ALL_CPUS</td>
<td>Return the identifiers of all platform CPUs that are available for I/O interrupts.</td>
</tr>
<tr>
<td>WSIO_MSI_NEXT_CPU</td>
<td>Return a single CPU identifier. If the caller passes in a CPU identifier in the cpu_array, the next CPU available, subject to the selection criteria, will be returned. If the caller passes in a CPU identifier of -1, the service will assign a CPU based on an internal round-robin algorithm. This hint is used mostly for MSI type interrupts to allow the platform code to round robin CPU assignments.</td>
</tr>
</tbody>
</table>

In case of the hints WSIO_MSI_CPU_LOCAL or WSIO_MSI_GET_ALL_CPUS, if the caller passes in a NULL value for the cpu_array parameter, wsio_msi_get_cpus will only update cnt to the number of CPUs that meet the selection criteria.

RETURN VALUES

WSIO_OK Successful completion. The identifiers of available CPUs are returned in the cpu_array.

WSIO_ERROR The system is not yet sufficiently initialized to provide a list of applicable CPUs, or an error occurred.

WSIO_INVALID_ISC The isc is not valid.

WSIO_INVALID_PARM The cpu_array is NULL (while hints is WSIO_MSI_NEXT_CPU), cnt is 0, or an invalid value for hints was specified.
CONSTRANTS
None

EXAMPLE
#define MAX_CPUS 64 /* Assume that the maximum number of CPUs is 64 */

wsio_ret_code_t ret;
wsio_cpu_id_t cpu_ids[MAX_CPUS];
int array_size, active_cpus, i;

array_size = MAX_CPUS;
ret = wsio_msi_get_cpus(isc, cpu_ids, &array_size, WSIO_MSI_GET_ALL_CPUS);
active_cpus = array_size;

for (i = 0; i < active_cpus; i++) {
    printf("cpuid == \%x\n", cpu_ids[i]);
}

SEE ALSO
wsio_msi_alloc(9F), wsio_msi_assign(9F), wsio_msi_capability(9F), wsio_msi_disable(9F),
wsi_msi_enable(9F), wsio_msi_free(9F), wsio_msi_query(9F), wsio_msi_resize(9F)
**NAME**

wsio_msi_query - Return per-vector attributes of an MSI/MSI-X interrupt object.

**SYNOPSIS**

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

wsio_ret_code_t wsio_msi_query(
    wsio_msi_hndl_t *hndl,
    wsio_msi_vec_info_t type,
    int first,
    int num,
    void *vec_info
);
```

**PARAMETERS**

- **hndl**: Handle of an MSI/MSI-X interrupt object.
- **type**: Type of vector information to return.
- **first**: Starting vector.
- **num**: Number of vectors to query.
- **vec_info**: Pointer to a buffer large enough to hold the number of specified vector attribute values. Its data type will vary depending on the value of the **type** parameter, so it must be typecast to a void pointer.

**DESCRIPTION**

The `wsio_msi_query` service is called to query a contiguous set of vector values of an MSI or MSI-X interrupt object.

Valid values for **type**, as well as the data type returned in the **vec_info** array, are shown in the following table:

**Table 2-10 Values for type**

<table>
<thead>
<tr>
<th><strong>wsio_msi_vec_info_t</strong></th>
<th><strong>type of vec_info</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_MSI_CPU_ID</td>
<td>wsio_cpu_id_t *</td>
<td>Return the identifier of the CPU that each vector’s interrupt is directed to. The value returned in vec_info will be an array of type wsio_cpu_id_t.</td>
</tr>
<tr>
<td>WSIO_MSI_ENABLED</td>
<td>int *</td>
<td>Return the status of the vector mask. A 0 value indicates the vector is enabled, non-zero indicates that it is disabled. The value returned in vec_info will be an array of integers.</td>
</tr>
<tr>
<td>WSIO_MSI_OBJ_TYPE</td>
<td>wsio_msi_type_t *</td>
<td>Return the type of interrupt object, either WSIO_MSI_TYPE or WSIO_MSI_X_TYPE. The value returned in vec_info will be a single value of type wsio_msi_type_t since an interrupt object can only have one type.</td>
</tr>
<tr>
<td>WSIO_MSI_PENDING</td>
<td>int *</td>
<td>Return the status of the pending bit. A 0 value indicates an interrupt is pending, non-zero indicates that it is not pending. The value returned in vec_info will be an array of integers. If an MSI function does not support per-vector masking, wsio_msi_query will return WSIO_NOT_SUPPORTED.</td>
</tr>
<tr>
<td>WSIO_MSI_VECTOR_CNT</td>
<td>int *</td>
<td>Return the number of vectors associated with the interrupt object. The value returned in vec_info will be a single integer.</td>
</tr>
</tbody>
</table>
RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR An error occurred.
WSIO_INVALID_PARM Invalid value for type, hndl, or vec_info, or invalid vector range specified by first and num parameters.
WSIO_NOT_SUPPORTED The platform does not have support for MSI/MSI-X features.

CONSTRAINTS

None

EXAMPLES

```c
wsio_ret_code_t ret;
wsio_msi_hndl_t msi_hndl;
spu_t vector_info[10];
wsio_msi_type_t object_type;

/* Retrieve the CPU identifiers of the first 10 vectors */
ret = wsio_msi_query(msi_hndl, WSIO_MSI_CPU_ID, 0, 10, (void *)vector_info);

/* Retrieve the type of the interrupt object */
ret = wsio_msi_query(msi_hndl, WSIO_MSI_OBJ_TYPE, 0, 1, (void *)&object_type);
```

SEE ALSO

wsio_msi_alloc(9F), wsio_msi_assign(9F), wsio_msi_capability(9F), wsio_msi_disable(9F),
wsio_msi_enable(9F), wsio_msi_free(9F), wsio_msi_get_cpus(9F), wsio_msi_resize(9F)
NAME

wsio_msi_resize - Resize the vector table of an MSI-X interrupt object.

SYNOPSIS

#include <sys/wsio.h>

wsio_ret_code_t wsio_msi_resize (
    wsio_msi_hndl_t *hndl,
    uint64_t *new_size
);

PARAMETERS

hndl Handle of an MSI-X interrupt object.
new_size Requested new size of the vector table. This will be updated to the actual allocated new size.

DESCRIPTION

The wsio_msi_resize service is called to change the number of vectors associated with an MSI-X type interrupt object. The parameter hndl is the handle of the interrupt object and new_size indicates the desired new size of the vector table. If the call is successful the actual number of vectors in the new object will be returned in new_size.

If the number of vectors is being increased, wsio_msi_resize will attempt to allocate at least as many vectors as requested and may round up the request to a larger size. If the requested number of vectors exceeds the number supported for the PCI device (as reported by wsio_msi_capability), wsio_msi_resize will return an error.

If the number of vectors is being decreased, wsio_msi_resize must not be called in an interrupt context and the vectors being removed must be disabled; otherwise, an error will be returned. An error will also be returned if the resize operation would remove all vectors from the object.

MSI type objects cannot be resized.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR An error occurred.
WSIO_INVALID_PARM The hndl is invalid, or new_size is NULL.
WSIO_NOT_SUPPORTED The platform does not have support for MSI/MSI-X features.

CONSTRAINTS

Do not call this service while holding a spinlock.

Do not called in an interrupt context, if the number of vectors is being decreased.

EXAMPLE

wsio_ret_code_t ret;
int vector_cnt, max_cnt;
wsio_msi_hndl_t msi_hndl;

/* Query the maximum number of MSI-X vectors for this device */
ret = wsio_msi_capability(my_isc, WSIO_MSI_X_CAPABILITY, &max_cnt);

/* Retrieve the current number of vectors in the object */
ret = wsio_msi_query(msi_hndl, WSIO_MSIVECTOR_CNT, 0, 0, &vector_cnt);
/* Increase the number of vectors by 5 */
vector_cnt += 5;

if (vector_cnt <= max_cnt) {
    /* Resize the object */
    ret = wsio_msi_resize(hndl, vector_cnt);

    if (ret == WSIO_OK) {
        printf("resized vector table to %d\n", vector_cnt);
    }
}

SEE ALSO

wsio_msi_alloc(9F), wsio_msi_assign(9F), wsio_msi_capability(9F), wsio_msi_disable(9F),
wsio_msi_enable(9F), wsio_msi_free(9F), wsio_msi_get_cpus(9F), wsio_msi_query(9F)
NAME

wsio_port_inXX - Read XX bits from an I/O port.

SYNOPSIS

#include <sys/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

None

EXAMPLE

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_iports(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(9F), wsio_map_port(9F), wsio_port_outXX(9F), wsio_unmap_port(9F)
NAME

wsio_port_outXX - Write XX bits to an I/O port.

SYNOPSIS

#include <sys/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

None

EXAMPLE

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(9F), wsio_map_port(9F), wsio_port_outXX(9F), wsio_unmap_port(9F)
wsio_probe_dev_info(9S)

NAME

wsio_probe_dev_info - WSIO device probe information.

SYNOPSIS

#include <sys/wsio.h>

struct wsio_probe_dev_info
{
    unsigned short instance;
    unsigned short target;
    unsigned short opt_1;
    unsigned short opt_2;
    unsigned short opt_3;
    unsigned short target;
};

DESCRIPTION

The wsio_probe_dev_info structure contains some of the device file information needed for
WSIO driver probe routines. This structure communicates hardware path information within
driver probe routines (especially for those cases where drivers may split their probe routines
into two routines; one for determining the next address to probe and one to actually try to build
and open a device file for that address). The hardware addressing information in this format can
easily be passed between two routines and is needed for building special device files for the
devices being probed.

STRUCTURE MEMBERS

instance Instance number of the nearest interface card ancestor.
target Relative hardware address of first layer to be probed.

opt_1 Optional. Sometimes used for the hardware address of the second layer to be
        probed (for example, LUN).

opt_2 Driver-discretionary element.

opt_3 Driver-discretionary element.

RETURN VALUES

CONTRAINS

None

EXAMPLE

A SCSI probe example might consist of two routines. The scsi_probe_function function
determines the next address to be probed and scsiProbe 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(9F)
NAME

wsio_query_supported_function - Return a pointer to a function supported by WSIO.

SYNOPSIS

#include <sys/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 WSIO function to register a driver’s event handler.
    WSIO_GET_HANDLER_UNREG_FUNC WSIO function to unregister a driver’s event handler.
    WSIO_GET_INSTALL_DRV_FUNC WSIO function to register a driver’s function.
    WSIO_GET_MASK_REG_FUNC 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 allows 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

None

EXAMPLE

static wsio_drv_info_t my_drv_info {
    ...
};
void my_event_handler()
{
    ...
}int my_install(void)
{
wsi_o_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) {

387
/* Now register our event handler with wsio */
ret = handler_reg_ptr(&my_drv_info,
    my_event_handler);
...
}
}
...

SEE ALSO

wsio_install_drv_event_handler(9F), wsio_install_drv_func(9F), wsio_reg_drv_capability_mask(9F),
wsio_uninstall_drv_event_handler(9F)
NAME

wsio_read_regXX - Read XX bits from a mapped device register.

SYNOPSIS

#include <sys/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 must 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. Therefore, if the local bus and the host bus are of opposite endianness, the driver must perform an endian swap.

RETURN VALUES

None

CONSTRAINTS

None

EXAMPLE

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(9F), wsio_map_reg(9F), wsio_unmap_reg(9F), wsio_write_regXX(9F)
NAME

wsio_reg_drv_capability_mask - Register a driver's capability mask.

SYNOPSIS

#include <sys/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 what operations the driver is capable of handling. The mask is an OR of wsio_events_t and is for an instance of the driver. This function must be called in a driver's attach routine after isc_claim is executed. This function can be called more than once to disable and enable certain functionalities. A new mask will supersede a previous one. If a platform does not support event handling, WSIO_HA_NA will be returned. In this case, a driver must ignore the error.

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_INVALID_EVENT Invalid event.

WSIO_INVALID_ISC Invalid isc.

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

CONSTRAINTS

None

EXAMPLE

my_driver_attach() {
    int ret;
    wsio_event_mask_t my_mask = WSIO_EVENT_SUSPEND |
        WSIO_EVENT_RESUME;
    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
    }
    run the rest of the attach chain
}

SEE ALSO

wsio_install_drv_event_handler(9F), wsio_query_supported_function(9E)
NAME

wsio_register_addr_probe - 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_register_addr_probe WSIO service 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 must 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 must
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 must 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
must return the additional information of probe_id, name and desc. The probe_id is a unique
identifier. The name string must 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
int mydrv_install()
{
    (void)wsio_register_addr_probe(mydrv_probe, "mydrv");
    return(wsio_install_driver(&mydrv_wsio_info));
}

CONSTRAINTS
SEE ALSO
drv_info(9S), wsio_drv_data_t(9S), wsio_register_dev_probe(9F)
wsio_register_dev_probe(9F)

NAME

wsio_register_dev_probe - 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 the driver data that the third parameter must match. 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_register_dev_probe WSIO service 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 must 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 wsio_register_dev_probe service must 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 must 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 must 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 must 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 must return the additional information of probe_id, name and desc. The probe_id is a unique identifier. The name string must match the drv_path field in the wsio_drv_data_t structure of the driver that controls the device. The desc parameter is an ASCII string describing the device.

RETURN VALUES
0 Successful completion.
-1 Error

CONSTRAINTS
None

EXAMPLE
int mydrv_install()
{
    (void)wsio_register_dev_probe(DRV_NAME, mydrvProbe, "mydrv");
    return(wsio_install_driver(&mydrv_wsio_info));
}

SEE ALSO
drv_info(9S), wsio_drv_data_t(9S), wsio_register_addr_probe(9F)
NAME

wsio_remap_dma_buffer - Map pre-allocated IOVAs to new host ranges.

SYNOPSIS

#include <sys/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.

The wsio_remap_dma_buffer function 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
None

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(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(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 does not 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(9F),wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_fastmap_dma_buffer(9F), wsio_free_dma_handle(9F), wsio_free_shared_mem(9F),
wsio_flush_shared_mem(9F), wsio_init_map_context(9F), wsio_iova_to_phys(9F),
wsio_map_dma_buffer(9F), wsio_set_device_attributes(9F), wsio_set_dma_attributes(9F),
wsio_unmap_dma_buffer(9F)
wsio_set_description(9F)

NAME

wsio_set_description - Set the I/O tree node description for this driver.

SYNOPSIS

#include <sys/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 must make a decision based upon whether the ioscan output for that device driver is understandable.

RETURN VALUES

None

CONSTRAINTS

None

EXAMPLE

wsio_set_description (isc_entry,"My Driver Description");

SEE ALSO
wsio_set_dma_attributes(9F)

NAME

wsio_set_dma_attributes - Associate DMA hints with a DMA handle.

SYNOPSIS

#include <sys/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
);

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 are:

<table>
<thead>
<tr>
<th>Attribute</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 Dependent</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 to flush the cacheline from any intermediate buffers as soon as it is
This inhibits any coalescing of data by bus bridges.
Default: 0

WSIO_DMA_ATTR_IGN_ALIGN
Specifies the mapping service must 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)
2 DMA buffers are static and accessed randomly (lowfat)
Default: 0

WSIO_DMA_ATTR_PREFETCH
Specifies how aggressively hardware must prefetch for outbound DMA.
0 No prefetch
1 Moderate prefetch
2 Aggressive prefetch
Default: 1

WSIO_DMA_ATTR_SAFE
Specifies the use of the most conservative coherency model 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 buses.
Default: Hardware dependent

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.
1 True for data buffers
param Driver-defined parameter passed as the first parameter to *isr*. Typically, *isc* is passed as *arg1*.

**DESCRIPTION**

The `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 `wsio_dma_set_device_attributes`, and are overridden by some hints passed in as parameters to `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 `wsio.allocate_shared_mem` or `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 `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**

None

**EXAMPLE**

```c
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(9F), wsio.allocate_shared_mem(9F), wsio_dma_pass_thru(9F),
wsio_dma_set_device_attributes(9F), wsio_fastmap_dma_buffer(9F), wsio_free_dma_handle(9F),
wsio_free_shared_mem(9F), wsio_flush_shared_mem(9F), wsio_init_map_context(9F),
wsio_iova_to_phys(9F), wsio_map_dma_buffer(9F), wsio_remap_dma_buffer(9F),
wsio_unmap_dma_buffer(9F)`
**NAME**

wsio_set_dma_callback - Set the callback function and argument for DMA.

**SYNOPSIS**

```c
#include <sys/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.

**CONSTRAINTS**

None

**EXAMPLE**

```c
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(9F)`, `wsio_allocate_shared_mem(9F)`, `wsio_dma_pass_thru(9F)`,
- `wsio_fastmap_dma_buffer(9F)`, `wsio_free_dma_handle(9F)`, `wsio_free_shared_mem(9F)`,
- `wsio_flush_shared_mem(9F)`, `wsio_init_map_context(9F)`, `wsio_map_dma_buffer(9F)`,
- `wsio_remap_dma_buffer(9F)`, `wsio_set_device_attributes(9F)`, `wsio_set_dma_attributes(9F)`,
- `wsio_unmap_dma_buffer(9F)`
NAME
wsio_sizeof_attribute - Returns the size of an attribute.

SYNOPSIS
#include <sys/wsio.h>
wsio_ret_code_t wsio_sizeof_attribute(
    IN struct isc_table_type *isc,
    IN char *name,
    OUT size_t *size
);

PARAMETERS
isc   The isc handle of the interface that the attribute is associated with.
name  The name of the attribute.
size  The size of the attribute.

DESCRIPTION
This service returns the size of an attribute identified by the name and isc parameters.

RETURN VALUES
WSIO_OK  The attribute size is returned.
WSIO_ERROR  An error occurred.

CONSTRAINTS
None

SEE ALSO
wsio_create_attribute(9F), wsio_modify_attribute(9F), wsio_get_attribute(9F), wsio_destroy_attribute(9F).
wsio_uninstall_driver(9F)

NAME

wsio_uninstall_driver - Uninstall a driver's header structure from the WSIO CDIO.

SYNOPSIS

#include <sys/wsio.h>

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. The wsio_uninstall_driver function is called by the driver prior to unloading.

RETURN VALUES

0        Successful completion.
<>0       Error

CONSTRAINTS

None

SEE ALSO

wsio_install_driver(9F)
NAME

wsio_uninstall_drv_event_handler - Uninstall a driver's event handler.

SYNOPSIS

#include <sys/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

None

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(9F)
NAME

wsio_unmap_cfg_handle - Release a configuration space handle.

SYNOPSIS

#include <sys/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

None

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(9F), wsio_cfg_inXX(9F), wsio_cfg_outXX(9F)
wsio_unmap_dma_buffer(9F)

NAME

wsio_unmap_dma_buffer - Remove a DMA packet mapping.

SYNOPSIS

#include <sys/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. The io_range must have been obtained via a previous call to wsio_map_dma_buffer, wsio_remap_dma_buffer, or wsio_fastmap_dma_buffer.

The wsio_unmap_dma_buffer function 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.

CONSTRAINTS

None

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(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 does not 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(9F), wsio_allocate_shared_mem(9F), wsio_dma_pass_thru(9F),  
wsio_fastmap_dma_buffer(9F), wsio_free_dma_handle(9F), wsio_free_shared_mem(9F),  
wsio_flush_shared_mem(9F), wsio_init_map_context(9F), wsio_iova_to_phys(9F),  
wsio_map_dma_buffer(9F), wsio_remap_dma_buffer(9F), wsio_set_device_attributes(9F),  
wsio_set_dma_attributes(9F)
wsio_unmap_port(9F)

NAME

wsio_unmap_port - Unmap an I/O port.

SYNOPSIS

#include <sys/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_ioports.
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 must not be accessed by the driver anymore.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR Parameter error.

CONSTRAINTS

None

EXAMPLE

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_get_ioports(9F), wsio_map_port(9F), wsio_port_inXX(9F), wsio_port_outXX(9F)
NAME

wsio_unmap_reg - Unmap a device register.

SYNOPSIS

#include <sys/wsio.h>

int wsio_unmap_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 unmapped. This must be the same structure used to map the register.

DESCRIPTION

The wsio_unmap_reg WSIO function removes the mapping done by wsio_map_reg.

RETURN VALUES

WSIO_OK Successful completion.
WSIO_ERROR Could not unmap the register.

CONSTRAINTS

None

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. */
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(9F), wsio_map_reg(9F), wsio_read_regXX(9F), wsio_write_regXX(9F)
NAME

wsio_unregister_dev_probe - Unregisters a driver probe function.

SYNOPSIS

#include <sys/wsio.h>

int wsio_unregister_dev_probe(
    int type,
    char *name
);

PARAMETERS

*type* Indicates the driver data that the second parameter must match. Valid values are:
- **IF_CLASS** The second argument, *name*, is to be matched with the *drv_path* field of the *wsio_drv_data_t* structure.
- **DRV_NAME** The second argument, *name*, is to be matched with the *name* field of the *drv_info_t* structure.

*name* An ASCII string indicating the name or class of the driver.

DESCRIPTION

The *wsio_unregister_dev_probe* WSIO service 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* must 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 must 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

0 Successfully found and deleted the driver.
-1 Not found.

CONSTRAINTS

None

EXAMPLE

```c
int mydrv_unload( void *arg)
{
    int ret;
    struct isc_table_type *isc;
    void *token, *priv_ptr;

    /****************************************************************************
    * Remove the attach function from the DLKM attach list
    ****************************************************************************/

    if (mod_wsio_attach_list_remove (MOD_WSIO_CORE,
```
&module_name_core_attach))
    return (ENXIO);

/****************************
 * Unregister the device probe
 ***************************/
(void) wsio_unregister_dev_probe(IF_CLASS, "mydrv_path");
    "probe_name";

/****************************
 * Uninstall the driver. If it fails, go back to the
 * load state and undo what has been done in the
 * unload routine.
 ***************************/
if(wsio_uninstall_driver(&module_name_wsio_info)) {
    return (ENXIO);
}
return(0);

SEE ALSO

wsio_drv_data_t(9S), wsio_drv_info(9S), wsio_register_dev_probe(9F)
wsio_write_regXX(9F)

NAME

wsio_write_regXX - Write XX bits to a mapped register.

SYNOPSIS

#include <sys/wsio.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 must 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. Therefore, if the local bus and the
host bus are of opposite endianness, the driver must perform an endian swap.

RETURN VALUES

None

CONSTRAINTS

None

EXAMPLE

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(9F), wsio_map_reg(9F), wsio_read_regXX(9F), wsio_unmap_reg(9F)
This chapter describes driver functions that are specific to PCI Services.

**PCI Macros and Functions**

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**

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(9F)*.
CONNECT_INIT_ROUTINE(9F)

NAME
CONNECT_INIT_ROUTINE - 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.
PCI_ATTACH_DEV_INIT_ERROR(9F)

NAME

PCI_ATTACH_DEV_INIT_ERROR - 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 must be used as appropriate in the device’s driver_attach or driver_if_init routine.
NAME

pci_desc_bus_transactions_isc - Describe the typical bus performance path transaction size.

SYNOPSIS

#include <sys/pci.h>

int pci_desc_bus_transactions_isc(
    struct isc_table_type *isc,
    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:

#define 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. A 0 value means do not change the current value.

write_width The number of 32-bit words in the width of the write path. A 0 value means do not 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 buses on the system. Also, performance depends on other buses 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 buses.

If you do not 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

None
NAME

pci_get_port_hndl_isc - 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(
    struct isc_table_type *isc,
    uint32_t *pci_io_addr,
    uint32_t size,
    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 is to 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. Therefore, call it only 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.

CONSTRAINTS

None

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.

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(9F)
NAME
pci_read_cfg_uintN_isc - Read unsigned integer from a PCI configuration register.

SYNOPSIS
#include <sys/pci.h>
void pci_read_cfg_uint8_isc(
    struct isc_table_type *isc,
    int reg_num,
    uint8_t *data_read
);
pci_read_cfg_uint16_isc(
    struct isc_table_type *isc,
    int reg_num,
    uint16_t *data_read
);
pci_read_cfg_uint32_isc(
    struct isc_table_type *isc,
    int reg_num,
    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 is to 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
None

EXAMPLE
#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(9F)
**NAME**

pci_read_port_uintN_isc - Read little-endian data from an I/O port.

**SYNOPSIS**

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

void pci_read_port_uint8_isc(
    struct isc_table_type *isc,
    PCI_PORT_HNDL ph,
    uint32_t offset,
    uint8_t *data
);

void pci_read_port_uint16_isc(
    struct isc_table_type *isc,
    PCI_PORT_HNDL ph,
    uint32_t offset,
    uint16_t *data
);

void pci_read_port_uint32_isc(
    struct isc_table_type *isc,
    PCI_PORT_HNDL ph,
    uint32_t offset,
    uint32_t *data
);
```

**PARAMETERS**

- `isc` Pointer to an ISC table associated with the device.
- `ph` A port handle previously obtained with a call to `pci_get_port_hndl_isc`.
- `offset` An offset from `ph`.
- `data` A pointer to an 8-, 16-, 32-bit location where the routine is to 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**

None

**EXAMPLES**

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

#define MY_IOMAP_BASE 0x10
#define MY_PORT_SIZE 0x100
#define MY_IDREG 0x0
#define MY_HWID 0x4850
static void mydriver_memset(struct isc_table_type *isc)
{
    unsigned short hwid;
```
unsigned int port_addr;
uint16_t old_cmdreg;
PCI_PORT_HNDL ph;

isc->mapped = NULL;
pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&old_cmdreg);
msg_printf("command reg = 0x%x\n",old_cmdreg);
pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND, old_cmdreg |
PCI_CMD_IO_SPACE | PCI_CMD_BUS_MASTER);
pci_read_cfg_uint32_isc(isc,MY_IOMAP_BASE,&port_addr);
port_addr &= ~3;
if (pci_get_port_hndl_isc(isc, port_addr,
MY_PORT_SIZE, &ph)) {
    pci_read_port_uint16_isc(isc,ph,MY_IDREG,&hwid);
    if (((hwid & MY_HWID) != MY_HWID) {        
        return -1;
    } else {        
        msg_printf("pci_get_port_hndl_isc() failed\n");        
        return -1;
    }
    return 0;
} else {
    msg_printf("pci_get_port_hndl_isc() failed\n");
    return -1;
}

SEE ALSO
pci_write_port_uintN_isc(9F)
NAME

pci_unget_port_hndl_isc - Delete a system defined handle for manipulating a range of PCI I/O ports.

SYNOPSIS

#include <sys/pci.h>

int pci_unget_port_hndl_isc(
    struct isc_table_type *isc,
    uint32_t pci_io_addr,
    uint32_t size,
    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

None

EXAMPLES

None

SEE ALSO

pci_get_port_hndl_isc(9F)
NAME

pci_write_cfg_uintN_isc - Write unsigned integer to a PCI configuration register.

SYNOPSIS

#include <sys/pci.h>

void pci_write_cfg_uint8_isc(
    struct isc_table_type *isc,
    int reg_num,
    uint8_t data_write
);

pci_write_cfg_uint16_isc(
    struct isc_table_type *isc,
    int reg_num,
    uint16_t data_write
);

pci_write_cfg_uint32_isc(
    struct isc_table_type *isc,
    int reg_num,
    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

None

SEE ALSO

pci_read_cfg_uintN_isc(9F)
NAME
pci_write_port_uintN_isc - Write little-endian data to an I/O port.

SYNOPSIS
#include <sys/pci.h>

void pci_write_port_uint8_isc
    (struct isc_table_type *isc,
     PCI_PORT_HNDL ph,
     uint32_t offset,
     uint8_t data );

void pci_write_port_uint16_isc
    (struct isc_table_type *isc,
     PCI_PORT_HNDL ph,
     uint32_t offset,
     uint16_t data );

void pci_write_port_uint32_isc
    (struct isc_table_type *isc,
     PCI_PORT_HNDL ph,
     uint32_t offset,
     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
None

SEE ALSO
pci_read_port_uintN_isc(9F)
READ_REG_UINTn_ISC(9F)

NAME

READ_REG_UINTn_ISC - Read and byte swap data from a little-endian bus.

SYNOPSIS

#include <sys/pci.h>

void READ_REG_UINT8_ISC(
    struct isc_table_type *isc,
    uint8_t *addr,
    uint8_t *data
);

#include <sys/pci.h>

void READ_REG_UINT16_ISC(
    struct isc_table_type *isc,
    uint16_t *addr,
    uint16_t *data
);

#include <sys/pci.h>

void READ_REG_UINT32_ISC(
    struct isc_table_type *isc,
    uint32_t *addr,
    uint32_t *data
);

PARAMETERS

isc  Pointer to an ISC table associated with the driver.

addr A pointer to the address of the bus data. It must be one of:

- A virtual address mapped with map_mem_to_host.
- A mapped offset in the automatically mapped first-base-address register range contained in isc->if_reg_ptr.
- Only the first nonzero 32-bit-wide memory base register found can be mapped, starting in the range 0x10 and searching up through 0x24 (the six possible base address register locations in configuration space).
- If that base-register's size is in excess of 8 KB, it is not mapped and isc->if_reg_ptr is set to NULL. In this case, the driver must map the base register it wants to use.
- The address of a memory buffer shared between the driver and a little-endian bus master.

data A pointer to an 8-, 16-, or 32-bit location where the routine it to 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 PCI_LITTLE_ENDIAN_ONLY flag 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
None

EXAMPLES
#include <sys/pci.h>
#define MYREGISTER_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(9F)
WRITE_REG_UINTn_ISC(9F)

NAME

WRITE_REG_UINTn_ISC - Byte swap and write data to a little-endian bus.

SYNOPSIS

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

void WRITE_REG_UINT8_ISC(
    struct isc_table_type *isc,
    uint8_t *addr,
    uint8_t data
);
#include <sys/pci.h>

void WRITE_REG_UINT16_ISC(
    struct isc_table_type *isc,
    uint16_t *addr,
    uint16_t data
);
#include <sys/pci.h>

void WRITE_REG_UINT32_ISC(
    struct isc_table_type *isc,
    uint32_t *addr,
    uint32_t data
);
```

PARAMETERS

- **isc**  
  Pointer to an ISC table associated with the device.
- **addr**  
  A pointer to the output address. It must be one of the following:
  - A virtual address mapped with `map_mem_to_host`.
  - A mapped offset in the automatically mapped first-base-address register range contained in `isc->if_reg_ptr`.
    - Only the first nonzero 32-bit wide memory base register found can be mapped, starting in the range 0x10 and searching up through 0x24 (the six possible base address register locations in configuration space).
    - If that base-register's size is in excess of 8 KB, it is not mapped and `isc->if_reg_ptr` is set to NULL. In this case, the driver must map the base register it wants to use.
  - The address of a memory buffer shared between the driver and a little-endian bus master.
- **data**  
  The 8-, 16-, or 32-bit data to be written.

DESCRIPTION

The WRITE_REG_UINTn_ISC PCI 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 `addr`.

If the PCI adapter that your card is running under has directly mapped the PCI memory space into driver-accessible system I/O space, you can improve the performance of WRITE_REG_UINTn_ISC if you define the `PCI_LITTLE_ENDIAN_ONLY` flag prior to including the `pci.h` header file. This causes WRITE_REG_UINTn_ISC to perform a simple byte swap instead of calling a function that tests byte ordering.
RETURN VALUES

The WRITE_REG_UINTn_ISC routines do not return values.

CONSTRAINTS

None

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(9F)
```
NAME

PCI_ERRATA-1 -

Memory Coherency Issues

Certain combinations of WSIO mapping service calls can interact with PCI masters on C class and J class processors to create an inconsistent view of memory.

It is possible for prefetching of host memory by the PA hardware chipsets to result in a PCI master reading stale data, even though the proper dma_sync calls have been made. The problem does NOT occur if:

- The PCI master does normal Memory Read transactions; the master does not master Memory Read Multiple (MRM) or Memory Read Line (MRL) transactions.
- The mapping is done with wsio_map with the IO_NO_SEQ and IO_SAFE flags set, regardless of the type of transactions the PCI master uses.
- The mapping is done with wsio_fastmap and the PCI master does NOT use MRM or MRL transactions.

DETAILS

There are two hardware prefetch buffers in the PA hardware chipset between memory and any PCI device. One is in the system's PCI bridge chip, and the other is system's GSC I/O bridge chip (which connects to the PA side of the PCI bridge chip). The I/O bridge chip has a cache line size prefetch buffer for each I/O (GSC) slot.

For the following discussion assume that an IO TLB was mapped using either wsio_fastmap or wsio_map without IO_NO_SEQ and IO_SAFE flag bits set.

When a PCI bus master runs an MRM or MRL transaction, the following events happen:

1. The PCI bridge chip requests a cache line (8 words), starting at the PCI master requested start address, from the I/O bridge chip. Because 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); for example, 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 (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(9F), wsio_fastmap(9F)`
NAME

PCI_ERRATA-2 -

PCI TRANSACTION ORDERING

Because of the 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, see the discussion on Transaction Ordering in the PCI Chapter of the *HP-UX 11i v3 Driver Development Guide*.

SEE ALSO
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:

- A device retries a configuration cycle.
- The device asserts DEVSEL and STOP during the same cycle.
- 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.
4 Network Device Driver Reference Pages

This chapter contains manual reference pages for the data structures, kernel and user space support routines, and macros essential for HP-UX network interface drivers.
NAME

DL_DISABMULTI_REQ - Request to disable multicast address.

SYNOPSIS

#include <sys/dlpi.h>

DESCRIPTION

This primitive is sent to drivers using the driver_controlp routine to disable a multicast address. The multicast address information is present in the dl_disabmulti_req_t structure.

The DL_DISABMULTI_REQ primitive described in this manpage may be different from the definition in DLPI Programmer's Guide. The information provided here is applicable only to tightly coupled drivers.

STRUCTURE MEMBERS

typedef struct {
    uint32_t    dl Primitive;
    uint32_t    dl_addr_length;
    uint32_t    dl_addr_offset;
} dl_disabmulti_req_t;

dl Primitive Set to DL_DISABMULTI_REQ.

dl_addr_length Length of the multicast address in bytes.

dl_addr_offset Offset from the beginning of the message block (MBLK->b_rptr) where the multicast address begins.

CONSTRAINTS

HP-DLPI will handle MAC and multicast addresses in canonical format for Ethernet drivers.

HP-DLPI will always succeed the DL_DISABMULTI_REQ request to the user/application, even if driver returned an error. If the driver fails the request for some reason, it will go out-of-sync with HP-DLPI and the driver can attempt to reprogram the multicast addresses (by getting them from HP-DLPI using a property) at a convenient time. Failure to do so, will result in performance degradation.

SEE ALSO

driver_controlp(9F)
NAME

DL_ENABMULTI_REQ - Request to enable multicast address.

SYNOPSIS

#include <sys/dlpi.h>

DESCRIPTION

This primitive is sent to drivers using the driver_controlp routine to enable a multicast address. The multicast address information is present in the dl_enabmulti_req_t structure.

The DL_ENABMULTI_REQ primitive described in this manpage may be different from the definition in DLPI Programmer’s Guide. The information provider here is applicable only to tightly coupled drivers.

STRUCTURE MEMBERS

typedef struct {
    uint32_t  dl_primitive;
    uint32_t  dl_addr_length;
    uint32_t  dl_addr_offset;
} dl_enabmulti_req_t;

dl_primitive Set to DL_ENABMULTI_REQ

dl_addr_length Length of multicast address in bytes.

dl_addr_offset Offset from the beginning of the message block (MBLK->b_rptr of the message block) where the multicast address begins.

CONSTRAINTS

HP-DLPI will handle MAC and multicast addresses in canonical format for Ethernet drivers.

HP-DLPI will send DL_ENABMULTI_REQ only for unique multicast addresses (if multiple streams enable same multicast address on the driver instance, only one DL_ENABMULTI_REQ will be sent).

HP-DLPI will automatically convert the DL_ENABMULTI_REQ to a multicast level, promiscuous-on request if the number of unique multicast addresses enabled on the interface exceeds the limit specified by the driver in dhc_max_mcast element of dl_hp_create_info_t. This will be done only if the driver instance is not already in multicast or higher promiscuous level.

SEE ALSO

driver_controlp(9F)
**NAME**

DL_GET_STATISTICS_REQ - Request to get the physical interface statistics.

**SYNOPSIS**

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

**DESCRIPTION**

This primitive is sent to the drivers using the `driver_controlp` routine to get the physical interface statistics. The associated structure for this request is `dl_get_statistics_req_t` structure.

The `DL_GET_STATISTICS_REQ` primitive described in this manpage may be different from the definition in DLPI Programmer’s Guide. The information provided here is applicable only to tightly coupled drivers.

**STRUCTURE MEMBERS**

```c
typedef struct {
    uint32_t dl_primitive;
} dl_get_statistics_req_t;
```

`dl_primitive` Set to `DL_GET_STATISTICS_REQ`.

**CONSTRAINTS**

The second MBLK (linked by `b_cont`) will contain space to hold the statistics of size `Ext_mib_t`. The second MBLK will be empty when it is passed to the driver from HP-DLPI (`b_rptr == b_wpwp`). After copying the statistics information to the second MBLK, the drivers must move the `b_wptr` to the size that was written to the second MBLK (in this case by the size of the `Ext_mib_t`). The driver can cast the `b_rptr` to be `Ext_mib_t *`, if it has to change some fields.

Drivers must not modify the `b_rptr`.

Even if driver supports only a subset of the statistics groups in `Ext_mib_t` structure, it must copy the complete structure.

`Ext_mib_t` is defined in `<sys/mib.h>`

**SEE ALSO**

`driver_controlp(9F)`
NAME

dl_hp_autoneg_sense_t - Auto Negotiation or Auto Sense value of the network interface.

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

This enumeration contains the values of Auto Negotiation or Auto Sense. This is used in dl_hp_op_param_t structure to set the driver instance auto-negotiation or auto-sense values.

STRUCTURE MEMBERS

typedef enum dl_hp_autoneg_sense {
    DL_HP_AUTONEG_SENSE_ON   = 0x1,
    DL_HP_AUTONEG_SENSE_OFF  = 0x2
} dl_hp_autoneg_sense_t;

DL_HP_AUTONEG_SENSE_ON       Auto Negotiation or Auto Sense is ON.
DL_HP_AUTONEG_SENSE_OFF      Auto Negotiation or Auto Sense is OFF.

SEE ALSO

dl_hp_op_param_t(9S)
dl_hp_create_info_t(9S)

NAME

dl_hp_create_info_t -- Structure to be passed during DL_HP_OP_CREATE operation.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

This data type used by tightly coupled or loosely coupled driver to register their instance with HP-DLPI. HP-DLPI will return the per-instance handle that identifies the HP-DLPI instance in later calls to HP-DLPI interfaces.

STRUCTURE MEMBERS

typedef struct dl_hp_create_info {
    uint32_t dhc_version;
    uint32_t dhc_instance_num;
    uint32_t dhc_ppa;
    dl_hp_mac_type_t dhc_mac_type;
    uint32_t dhc_mac_addr_len;
    uint32_t dhc_nm_id;
    uint32_t dhc_mtu;
    uint32_t dhc_major_num;
    uint32_t dhc_max_mcast;
    uint32_t dhc_max_vlan;
    dl_hp_drv_features_one_t dhc_features_one;
    uint32_t dhc_features_two;
    uint32_t dhc_features_three;
    dl_hp_encaps_type_t dhc_encaps;
    uint8_t dhc_mac_addr[DL_HP_MAX_MAC_ADDR_LEN];
    char dhc_hw_path[DL_HP_MAX_HDW_PATH_LEN];
    char dhc_drv_name[DL_HP_MAX_DRIVER_NAME_LEN];
    char dhc_arpmod_name[DL_HP_MAX_ARPMOD_NAME_LEN];
    uint32_t (*dhc_outputp)(void *drv_output_hdlp, mblk_t *mblkp, dl_hp_pkt_type_t pkt_type, void *rsvdp);
    uint32_t (*dhc_build_hdrp)(void *drv_output_hdlp, dl_hp_src_llc_info_t *src_infop, dl_hp_dest_llc_info_t *dset_infop, dl_hp_llc_info_t *hdr_infop, void *rsvd1p, void *rsvd2p);
    void (*dhc_controlp)(void *drv_control_hdlp, int32_t cmd, mblk_t *mblkp, void *rsvdp);
    int32_t (*dhc_eventp)(void *drv_event_hdlp, dl_hp_drv_event_type_t event_type, void *event_infop, void *rsvdp);
    int32_t (*dhc_drv_propp)(void *drv_prop_hdlp, uint32_t opcode, uint32_t prop_name, void *valuep, void *rsvd1p, void *rsvd2p);
    void *dhc_output_hdlp;
    void *dhc_control_hdlp;
    void *dhc_event_hdlp;
    void *dhc_mod_hdlp;
} dl_hp_create_info_t;
void *dhc_dlpi_hdlp;
void *dhc_reserved[3];
}
dl_hp_create_info_t

dhc_arpmod_name

From driver to HP-DLPI.
Character string specifying the ARP (STREAMS) module name for this driver, if one is required.
The name must be less than or equal to DL_HP_MAX_ARPMOD_NAME_LEN characters and must be NULL terminated. The DL_HP_MAX_ARPMOD_NAME_LEN constant is defined as 64.
Set by drivers only during registration of the interface.

dhc_build_hdrp

From driver to HP-DLPI.
The driver routine to build the MAC/LLC header in the outbound path. See driver_build_hdrp(9F) for details.
Set by physical drivers only during registration of the driver instance.
The driver can set this value to NULL if it does not have a build header routine. In this case, HP-DLPI uses its internal build header routine for Ethernet drivers.

dhc_control_hdlp

From driver to HP-DLPI.
The driver specified opaque (to HP-DLPI) handle that is passed to the driver control request processing routine.

dhc_controlp

From driver to HP-DLPI.
The driver routine for all control requests. See driver_controlp(9F) for more information.
Set by drivers only during registration of the driver instance.

dhc_dlpi_hdlp

From HP-DLPI to the driver.
Opaque handle that drivers must use in future call to HP-DLPI interfaces.

dhc_drv_name

From driver to HP-DLPI.
Driver name as a (null terminated) character string. The name must be less than or equal to DL_HP_MAX_DRIVER_NAME_LEN characters and must be NULL terminated. The DL_HP_MAX_DRIVER_NAME_LEN constant is defined as 64.
Set by drivers only during registration of the interface.

dhc_drv_propp

From driver to HP-DLPI.
Driver's property exchange routine. DLPI does not call this driver interface.
This information is optional and the field must be set to NULL if it is not implemented.

dhc_encaps

From driver to HP-DLPI.
Specifies the encapsulations supported by the drivers. For supported encapsulation types, refer to dl_hp_encaps_type_t.
Set by physical drivers only during registration of the interface.
The field is displayed by the lanscan command.
dhc_event_hdlp From driver to HP-DLPI
The driver specified opaque (to HP-DLPI) handle that is passed to the driver event handler.

dhc_eventp From driver to HP-DLPI.
The driver event handler to handle events defined by the DLS provider. See driver_eventp(9F) for more information.
Set by drivers only during registration of the driver instance.

dhc_features_one From driver to HP-DLPI.
See dl_hp_features_one_t(9S) for the features supported and the explanation of each one of the features.
Set by drivers during registration and can be changed using DLPI_PROP_FEATURES_ONE.

dhc_features_two From driver to HP-DLPI.
Reserved for future use. Set to 0.

dhc_features_three From driver to HP-DLPI.
Reserved for future use. Set to 0.

dhc_hdw_path From driver to HP-DLPI.
The hardware path of the physical interface as a (null terminated) character string. The length of the hardware path could be a maximum of DL_HP_MAX_HDW_PATH_LEN bytes, including NULL termination. The DL_HP_MAX_HDW_PATH_LEN constant is defined as 100.
Logical drivers can use the hardware path to specify a unique string for each instance of the logical driver.
Set by drivers only during registration of the interface.
The field is displayed by the lanscan command.

dhc_instance_num From driver to HP-DLPI.
An identifier that maps one-to-one with a physical interface belonging to a particular class. Set by drivers only during registration of the interface. The field is displayed by the lanscan command.

dhc_mac_addr_len From driver to HP-DLPI.
Length in bytes of the MAC address. Set by physical drivers only during registration of the interface.

dhc_mac_addr From driver to HP-DLPI.
The MAC address of the card, assumed to be dhc_mac_addr_len bytes in length and a maximum of DL_HP_MAX_MAC_ADDR_LEN bytes in length. The DL_HP_MAX_MAC_ADDR_LEN constant is defined as 32.
The MAC address must be in canonical format for Ethernet and FDDI drivers and in wire format for TR drivers.
Most drivers may not be able to provide the MAC address at the time of registering the instance. Hence, drivers need to explicitly set the MAC address (via DLPI_PROP_MAC_ADDRESS property)
after it is available. The drivers which are unable to set the MAC addresses during create must set this value to 0.

**dhc_mac_type**

From driver to HP-DLPI.

Set by drivers only during registration of the interface. The field is displayed by the `lanscan` command.

**dhc_major_num**

From driver to HP-DLPI.

This field is valid only for loosely coupled drivers. For tightly coupled drivers HP-DLPI sets this value to the HP-DLPI major number.

This field must be set to the major number of the DLS module used by the driver. Set by drivers only during registration of the interface.

Field is displayed by the `lanscan` command.

**dhc_max_mcast**

From driver to HP-DLPI.

The driver specified maximum number of unique multicast addresses supported by the driver/NIC in the range 1-0x7FFFFFFF.

Set by physical drivers only during registration of the driver instance and cannot be modified thereafter.

Drivers that do not have any such limits (such as those that use imperfect filtering) must set this field to 0xFFFFFFFF.

**dhc_max_vlan**

From driver to HP-DLPI.

The driver specified maximum number of VLANs supported by the driver or NIC in the range 0-4094.

Drivers can set this only during registration of the driver instance; it cannot be modified at a later time.

**dhc_mtu**

From driver to HP-DLPI.

The default MTU for the link in bytes. The minimum MTU must be 10 and the minimum recommended MTU is 256.

Set by drivers during registration of the interface. Can be modified anytime later (via the `DLPI_PROP_OP_PARAM` property)

**dhc_mod_hdlp**

Reserved for future use. Must be set to NULL.

**dhc_nm_id**

From driver to HP-DLPI.

The network management ID. Set by loosely coupled drivers during registration of the interface and can be updated later via the `DLPI_PROP_NMID` property.

This field is ignored for tightly coupled drivers as the HP-DLPI provider sets this property for an interface instance and provides it to drivers as a “get” property (DLPI_PROP_NMID). The DLPI_PROP_NMID property must be called by tightly coupled drivers if the NMID needs to be switched (atomically) for high availability applications (especially with IPv6). Tightly coupled drivers will be notified of the new NMID via the event handler interface (DL_HP_DRV_EVENT_CHANGE_NMID event).

Loosely coupled drivers must provide this value during registration and if the NMID changes during the course of operation, they must use DLPI_PROP_NMID “set” property to update the value in DLPI.
dhc_output_hdlp  From driver to HP-DLPI.
The driver specified opaque (to HP-DLPI) handle that is passed to
the outbound routines.

dhc_outputp  From driver to HP-DLPI.
The driver outbound routine. See driver_outputp(9F) for information.
Set by drivers only during registration of the driver instance.

dhc_ppa  From driver to HP-DLPI.
PPA of the interface. Valid only for loosely coupled drivers. For
tightly coupled drivers, the dhc_instance_num will be used as
PPA.

dhc_reserved  Reserved for future use. Must be set to 0.
dhc_version  From driver to HP-DLPI.
HP-DLPI Version that the driver can work with. Set by drivers only
during registration of the interface.

SEE ALSO
dlpi_propp(9F), driver_controlp(9F), driver_eventp(9F), dl_hp_encaps_type_t(9S),
dl_hp_drv_features_one_t(9S), lanscan(1M), driver_build_hdrp(9F)
NAME
dl_hp_dest_llc_info_t - Exchange destination information during LLC build header routine.

SYNOPSIS
#include <sio/dlpi_drz.h>

DESCRIPTION
This structure is used to pass destination information to the LLC build header routine from the
DLS provider before the call to the driver handle is made.

STRUCTURE MEMBERS

typedef struct dl_hp_dest_llc_info {
    uint32_t      ddli_dest_addr_length;
    uint8_t       ddli_dest_addr[DL_HP_MAX_DLSAP_LEN];
    uint64_t      ddli_reserved1[2];
    uint64_t      ddli_reserved2[2];
} dl_hp_dest_llc_info_t;

ddli_dlsap_addr    Identifies the destination DLSAP address. See the DLPI
Programmer’s Guide for more information on supported DLSAP formats.

ddli_dlsap_addr_length Identifies the DLSAP address length in bytes.

ddli_reserved1     Reserved for future use. It must be set to 0.

ddli_reserved2     Reserved for future use. It must be set to 0.

SEE ALSO
dl_hp_drv_features_one_t(9S)

NAME
dl_hp_drv_features_one_t - Driver features.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
Enumeration of driver features. These are bit-wise flags. Each feature consumes a bit in the
dhc_drv_features_one field passed to dlpi_propp. Drivers may need to use a combination of
these bits to specify their capabilities.

STRUCTURE MEMBERS
typedef enum dl_hp_drv_features_one {
    DL_HP_DRV_HP_DLS,
    DL_HP_DRV_IPV4CKO_IN,
    DL_HP_DRV_IPV4CKO_OUT,
    DL_HP_DRV_IPV6CKO_IN,
    DL_HP_DRV_IPV6CKO_OUT,
    DL_HP_DRV_COW,
    DL_HP_DRV_LFP,
    DL_HP_DRV_LNP,
    DL_HP_DRV_NO_SRC_ROUTING,
    DL_HP_DRV_NO_FASTPATH,
    DL_HP_DRV_HP_APA,
    DL_HP_DRV_VLAN,
    DL_HP_DRV_VTO,
    DL_HP_DRV_VI,
    DL_HP_DRV_PHYSICAL,
    DL_HP_DRV_LINKAGG,
    DL_HP_DRV_XPORT_OFFLOAD,
    DL_HP_DRV_LAN_CLASS,
    DL_HP_DRV_NOMULTIFRAGCKO,
    DL_HP_DRV_ILAN,
    DL_HP_DRV_TRAIN,
    DL_HP_DRV_64BIT_MIB,
    DL_HP_DRV_MGMT_INSTANCE
} dl_hp_drv_features_one_t;

DL_HP_DRV_HP_DLS
Setting this feature bit says that the driver is a tightly
coupled driver. Loosely coupled drivers must not set this
feature bit. Loosely coupled drivers are limited to using
dlpi_propp to:
• Register the interface.
• Change certain fields like MAC address, hardware
  state for display via lanscan.
• Set operational parameter MTU.
Other HP-DLPI interfaces (inbound routine, event handler,
and so on) must not be used by loosely coupled drivers.
Except DL_HP_DRV_LAN_CLASS, and
DL_HP_DRV_64BIT_MIB features, no other feature bits
must be set.
HP-DLPI will not allow streams to attach to the interfaces that do not have this features set.

**DL_HP_DRV_IPV4CKO_IN**

Driver supports inbound IPv4 CKO. The drivers setting this feature bit must get the inbound OOP template from HP-DLPI using the `DLPI_PROP_OOP_INFOP` property. This feature may be enabled/disabled after the initialization. Any change in feature must trigger an associated link DOWN and UP events.

The change in this feature must also be followed by a `DLPI_PROP_OOP_INFOP` property. See the *HP-UX 11i v2 Driver Development Guide* for more information.

**DL_HP_DRV_IPV4CKO_OUT**

Driver supports outbound IPv4 CKO. The drivers setting this feature bit must get the inbound OOP template from HP-DLPI using the `DLPI_PROP_OOP_INFOP` property. This feature may be enabled/disabled after the initialization. Any change in feature must trigger an associated link DOWN and UP events.

The change in this feature must also be followed by a `DLPI_PROP_OOP_INFOP` property. See the *HP-UX 11i v2 Driver Development Guide* for more information.

**DL_HP_DRV_IPV6CKO_IN**

For drivers that support inbound IPv6 CKO. The drivers setting this feature bit must get the inbound OOP template from HP-DLPI using the `DLPI_PROP_OOP_INFOP` property. This feature may be enabled/disabled after the initialization. Any change in feature must trigger an associated link DOWN and UP events.

The change in this feature must also be followed by a `DLPI_PROP_OOP_INFOP` property. See the *HP-UX 11i v2 Driver Development Guide* for more information.

**DL_HP_DRV_IPV6CKO_OUT**

For drivers that support outbound IPv6 CKO. The drivers setting this feature bit must get the inbound OOP template from HP-DLPI using the `DLPI_PROP_OOP_INFOP` property. This feature may be enabled/disabled after the initialization. Any change in feature must trigger an associated link DOWN and UP events.

The change in this feature must also be followed by a `DLPI_PROP_OOP_INFOP` property. See the *HP-UX 11i v2 Driver Development Guide* for more information.

**DL_HP_DRV_COW**

Reserved for internal use.

**DL_HP_DRV_LFP**

Reserved for internal use.

**DL_HP_DRV_LNP**

Reserved for internal use.

**DL_HP_DRV_NO_SRC_ROUTING**

Reserved for internal use.

**DL_HP_DRV_NO_FASTPATH**

This flag must be set if the drivers does not support fast-path. See the *HP-UX 11i v2 Driver Development Guide* for more information.

**DL_HP_DRV_HP_APA**

Reserved for internal use.

**DL_HP_DRV_VLAN**

Driver supports VLANs.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_DRV_VTO</td>
<td>Drivers support VLAN Tag Offload. Drivers insert VLAN tags (on outbound frames) and extract VLAN tags (on inbound frames).</td>
</tr>
<tr>
<td>DL_HP_DRV_VI</td>
<td>Reserved for internal use.</td>
</tr>
<tr>
<td>DL_HP_DRV_PHYSICAL</td>
<td>Set if the interface is a physical interface.</td>
</tr>
<tr>
<td>DL_HP_DRV_LINKAGG</td>
<td>Reserved for internal use.</td>
</tr>
<tr>
<td>DL_HP_DRV_XPORT_OFFLOAD</td>
<td>Reserved for internal use.</td>
</tr>
<tr>
<td>DL_HP_DRV_LAN_CLASS</td>
<td>Physical drivers register their class with the I/O subsystem. The instance numbers (passed in dhc_instance_num field of the dl_hp_create_info_t structure) are unique only within a driver class. All tightly coupled drivers must specify the lan class. Hence, all physical drivers that set DL_HP_DRV_HP_DLS must set this bit. Loosely coupled drivers may or may not specify the lan class, during the registration with I/O subsystem. If they do belong to lan class, it implies that these driver (and their native DLPI) support LAN commands (lanadmin and lanscan) and their instance numbers are in the range allocated for lan class drivers. Such drivers must also set the DL_HP_DRV_LAN_CLASS bit in the features field during registration with the HP-DLPI. However, as these drivers are loosely coupled, the DL_HP_DRV_HP_DLS bit must not be set.</td>
</tr>
<tr>
<td>DL_HP_DRV_NOMULTIFRAGCKO</td>
<td>Driver does not support CKO for multi-fragmented packets. Setting this bit without any of the DL_HP_DRV_CKO* features bit is invalid. This flag must be set only if DL_HP_DRV_TRAIN is also set. This feature may be enabled/disabled after the initialization. Any change in feature must trigger an associated link DOWN and UP events. See the HP-UX 11i v2 Driver Development Guide for more information.</td>
</tr>
<tr>
<td>DL_HP_DRV_ILAN</td>
<td>Reserved for internal use.</td>
</tr>
<tr>
<td>DL_HP_DRV_TRAIN</td>
<td>Driver supports packet trains.</td>
</tr>
<tr>
<td>DL_HP_DRV_64BIT_MIB</td>
<td>Drivers must set this bit during instance registration if the interface supports 64-bit MIB statistics. However, drivers can reset this feature bit anytime, if needed.</td>
</tr>
<tr>
<td>DL_HP_DRV_MGMT_INSTANCE</td>
<td>Reserved for internal use.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

dlpi_propp(9F), dl_hp_create_info_t(9S), dl_hp_oop_type_t(9S)
**NAME**

dl_hp_drv_param_req_type_t - Sub-request type for DL_HP_SET_DRV_PARAM_IOCTL and DL_HP_GET_DRV_PARAM_IOCTL.

**SYNOPSIS**

```
#include <sio/dlpi_drv.h>
```

**DESCRIPTION**

This structure provides an enumeration for the sub-request type that is used in the transparent IOCTLs defined by HP-DLPI.

**STRUCTURE MEMBERS**

```c
typedef enum dl_hp_drv_param_req_type {
    DL_HP_DRV_SPEED = 0x1,
    DL_HP_DRV_DUPLEX = 0x2,
    DL_HP_DRV_AUTONEG = 0x4,
    DL_HP_DRV_MTU = 0x8,
    DL_HP_DRV_RESET_MTU = 0x10,
    DL_HP_VALUE1 = 0x20,
    DL_HP_VALUE2 = 0x40,
    DL_HP_VALUE3 = 0x80,
    DL_HP_RESERVED1 = 0x100,
    DL_HP_RESERVED2 = 0x200,
    DL_HP_RESERVED3 = 0x400,
    DL_HP_RESERVED4 = 0x800,
    DL_HP_SERIALIZE = 0x40000000
} dl_hp_drv_param_req_type_t;
```

- **DL_HP_DRV_SPEED**
  Get or Set speed of the interface. The `dl_speed` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` must be used to get or set the driver speed.

- **DL_HP_DRV_DUPLEX**
  Get or Set Duplex mode of the interface. The `dl_duplex` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` must be used to get or set the driver duplex mode.

- **DL_HP_DRV_AUTONEG**
  Get or Set the Autonegotiation or Autosense value of the interface. The `dl_autoneg` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` must be used to get or set the driver's autonegotiation or autosense value.

- **DL_HP_DRV_MTU**
  Get or Set the MTU of the interface. The `dl_mtu` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` must be used to get or set the driver MTU.

- **DL_HP_DRV_RESET_MTU**
  Reset the MTU of the interface to its default value.

- **DL_HP_VALUE1**
  Driver-specific extension. The `dl_value1` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` may be used to get or set a driver property.
Driver-specific extension. The `dl_value2` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` may be used to get or set a driver property.

Driver-specific extension. The `dl_value3` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` may be used to get or set a driver property.

Driver-specific extension. The `dl_reserved1[0]` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` may be used to get or set a driver property.

Driver-specific extension. The `dl_reserved1[1]` field of `dl_hp_get_drv_param_ioctl_t` or `dl_hp_set_drv_param_ioctl_t` may be used to get or set a driver property.

Reserved for internal use by HP-DLPI.

Reserved for internal use by HP-DLPI.

Request HP-DLPI to serialize the request. Only applicable to `DL_HP_SET_DRV_PARAM_IOCTL`.

SEE ALSO

`DL_HP_SET_DRV_PARAM_IOCTL(9G)`, `DL_HP_GET_DRV_PARAM_IOCTL(9G)`
NAME

dl_hp_duplex_mode_t - Duplex setting of the interface.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

Duplex setting of the interface. It is used in the dl_hp_op_param_t structure to set the duplex setting of the network interface in HP-DLPI.

STRUCTURE MEMBERS

typedef enum dl_hp_duplex_mode {
    DL_HP_HALF_DUPLEX,
    DL_HP_FULL_DUPLEX
} dl_hp_duplex_mode_t;

DL_HP_HALF_DUPLEX Network interface is in half duplex mode.
DL_HP_FULL_DUPLEX Network interface is in full duplex mode.

SEE ALSO

dlpi_propp(9F), dl_hp_op_param_t(9S)
dl_hp_encaps_type_t(9S)

NAME

dl_hp_encaps_type_t - Enumerate data type for packet encapsulation types.

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

Encapsulation types supported by the interface

STRUCTURE MEMBERS

typedef enum {
    DL_HP_IEEE               = 0x01,
    DL_HP_EXT_IEEE           = 0x02,
    DL_HP_SNAP               = 0x04,
    DL_HP_ETHERTYPE          = 0x08,
    DL_HP_NOVELL             = 0x10
} dl_hp_encaps_type_t;

DL_HP_IEEE     IEEE 802.3 encapsulation.
DL_HP_EXT_IEEE  IEEE 802.3 extended SAP encapsulation type.
DL_HP_SNAP     SNAP 802.3 encapsulation type.
DL_HP_ETHERTYPE Ether encapsulation type.
DL_HP_NOVELL   Novell encapsulation type.

SEE ALSO

dlpi_propp(9F)
NAME
dl_hp_event_link_cause_t - Cause for driver events.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
This structure provides an enumeration of hints sent using the event_infop parameter for DL_HP_EVENT_INTERFACE_UP and DL_HP_EVENT_INTERFACE_DOWN events.

HP-DLPI does not in any way use the following hints, but may pass the information to other interested modules. Hence, drivers are still expected to notify HP-DLPI of the events such as, a change in IfOper/IfAdmin status, hardware state, link speed, MTU, MAC address, driver features and so on, explicitly using the dlpi_propp property exchange mechanism.

STRUCTURE MEMBERS
typedef enum dl_hp_event_link_cause {
    DL_HP_EVENT_USER_RESET,
    DL_HP_EVENT_DRIVER_RESET,
    DL_HP_EVENT_OP_PARAM_CHANGE,
    DL_HP_EVENT_ADDR_CHANGE,
    DL_HP_EVENT_PCI_ERR,
    DL_HP_EVENT_PARENT_DOWN,
    DL_HP_EVENT_MEMBERS_DOWN,
    DL_HP_EVENT_CABLE,
    DL_HP_EVENT_HW_STATE_DOWN,
    DL_HP_EVENT_IFADMIN_DOWN,
    DL_HP_EVENT_FEATURES_CHANGE,
    DL_HP_EVENT_HA,
    DL_HP_EVENT_DEFAULT
} dl_hp_event_link_cause_t;

DL_HP_EVENT_USER_RESET This hint is sent by the driver if the reset of the interface occurred due to the DL_HP_HW_RESET primitive.

DL_HP_EVENT_DRIVER_RESET This hint is sent by the driver if the reset is internally generated by the card/driver. This could occur due to a DMA timeout.

DL_HP_EVENT_OP_PARAM_CHANGE This hint is associated with change of an operational parameter for an interface.

DL_HP_EVENT_ADDR_CHANGE This hint is associated with a MAC address change.

DL_HP_EVENT_CABLE This hint is associated with a cable disconnect or reconnect.

DL_HP_EVENT_HW_STATE_DOWN This hint is associated with the NIC going to a dead state.

DL_HP_EVENT_IFADMIN_DOWN This hint is associated with ifAdmin change. For a list of supported ifadmin values, see dl_hp_ifadmin_state_t(9F).

DL_HP_EVENT_FEATURES_CHANGE This hint is associated with driver features change.

DL_HP_EVENT_HA This hint is currently unused. It is reserved for future use.

DL_HP_EVENT_DEFAULT This is the default event hint. Can be used for link UP events when driver does not know the exact cause.
SEE ALSO

dl_hp_ifadmin_state_t(9S)
dl_hp_event_type_t(9S)

NAME

dl_hp_event_type_t - Driver events.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

This is a list of events that can be passed from the driver to the HP-DLPI event handler. The enumerated type defines the driver events that are supported by HP-DLPI. Each event is represented as a bit flag.

Some events have associated information passed through the event_infop parameter. For example, in the case of DL_HP_EVENT_INTERFACE_DOWN and DL_HP_EVENT_INTERFACE_UP events, the cause field is provided by the driver. The cause field is sent as the event_infop parameter of the event handler for link up or down events.

STRUCTURE MEMBERS

typedef enum dl_hp_event_type {
    DL_HP_EVENT_INTERFACE_DOWN,
    DL_HP_EVENT_INTERFACE_UP,
    DL_HP_EVENT_OLD,
    DL_HP_EVENT_UNLOAD_CHECK,
    DL_HP_EVENT_CLEAR_STATS,
    DL_HP_EVENT_ARP_XMIT_REQ
} dl_hp_event_type_t;

DL_HP_EVENT_INTERFACE_DOWN This event must be sent whenever the interface enters a state that disallows data transfer. That can occur due to any one of the following:

• Physical link went down.
• ifAdmin status changed to DOWN on the interface.
• Card being reset/suspended/dead.
• The driver detecting an error and thus transitioning to a state indicating error.
• Change in driver features.
• Change in MAC address and other operational parameters.

See dl_hp_event_link_cause_t(9F) for further information on filling the event_infop parameter.

DL_HP_EVENT_INTERFACE_UP This event could occur due to any one of the following reasons:

• The physical link reconnected/recovered from an error state/resumed.
• The ifAdmin status changed to UP on the interface.
• The card reset is complete.
• Change in driver features.
• Change in MAC address and other operational parameters.
This event will be instrumental in informing the upper layers about change in the interface properties they have cached.

See `dl_hp_event_link_cause_t(9F)` for further information on filling the `event_infop` parameter.

**SEE ALSO**

`dl_hp_event_link_cause_t(9S)`

---

**DL_HP_EVENT_OLD**

This is not supported.

**DL_HP_EVENT_UNLOAD_CHECK**

This is not supported.

**DL_HP_EVENT_CLEAR_STATS**

This request must be issued by the driver after it clears its MIB statistics.

This event is reserved for internal use only.
NAME

DL_HP_GET_DRV_PARAM_IOCTL - HP-DLPI defined transparent IOCTL to get driver parameters.

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

This IOCTL is sent to drivers using the driver_controlp routine. The associated structure for this request is dl_hp_get_drv_param_ioctl_t structure.

This IOCTL is defined by HP-DLPI for usage by drivers. This IOCTL is transparent to HP-DLPI. Transparent IOCTLs are the ones that are not interpreted by HP-DLPI and are sent directly to the driver.

This IOCTL can be used to get driver operating parameters like Speed, Duplicity, MTU and Autonegotiation. Drivers can add new sub-requests to satisfy any driver specific commands.

STRUCTURE MEMBERS

typedef struct dl_hp_get_drv_param_ioctl {
    dl_hp_drv_param_req_type_t  dl_request;
    uint32_t                    dl_speed;
    dl_hp_duplex_mode_t         dl_duplex;
    dl_hp_autoneg_sense_t       dl_autoneg;
    uint32_t                    dl_mtu;
    uint64_t                    dl_value1;
    int64_t                     dl_value2;
    uint64_t                    dl_value3;
    uint32_t                    dl_reserved1[2];
    uint32_t                    dl_reserved2[2];
} dl_hp_get_drv_param_ioctl_t;


dl_request  The sub-request. The driver properties that need to be retrieved (see dl_hp_drv_param_req_type_t(9F)). The caller can set multiple requests in dl_request. Depending on the requests set in dl_request, the driver may return values in one or more of the following fields.

dl_speed    Speed of underlying link in Mbits/sec.

dl_duplex   Duplex modes. The field is of type dl_hp_duplex_mode_t.

dl_autoneg  Auto-negotiation/auto-sense. The field is of type dl_hp_autoneg_sense_t.

dl_mtu      MTU of the interface in octets.

dl_value1   64 bit value for driver extension.

dl_value2   64 bit value for driver extension.

dl_value3   64 bit value for driver extension.


dl_reserved2[2]  Reserved for internal use by HP-DLPI. Must be set to 0.

CONSTRAINTS

HP-DLPI does not interpret this request. This request must be used between the driver's user space application and the driver.
The DL_HP_SERIALIZE bit in dl_hp_drv_param_request_type_t is ignored for DL_HP_GET_DRV_PARAM_IOCTL.

The drivers can define their own dl_request by using any other bits that are not defined by HP-DLPI.

It is driver's responsibility to check the privileges of the user who issued the request (using ioc_cr field of iocblk).

The driver must fill up the ioc_count value in the IOCTL MBLK if the request is successfully processed.

**SEE ALSO**

dlpi_controlp(9F), dl_hp_drv_param_req_type_t(9S), dl_hp_duplex_mode_t(9S),
dl_hp_autoneg_sense_t(9S)
NAME
DL_HP_GET_MIBSTATS_REQ - Request to get the physical interface statistics.

SYNOPSIS
#include <sys/dlpi_ext.h>

DESCRIPTION
This primitive is sent to drivers using the driver_controlp routine to get the physical interface statistics. The associated structure for this request is dl_hp_get_mibstats_req_t structure. This primitive only applicable between HP-DLPI and the driver.

STRUCTURE MEMBERS
typedef struct {
    uint32_t dl_primitive;
    uint32_t dl_reserved1;
    uint32_t dl_reserved2;
} dl_hp_get_mibstats_req_t;

dl_primitive Set to DL_HP_GET_MIBSTATS_REQ

dl_reserved1 Reserved for future use. Set to 0.

dl_reserved2 Reserved for future use. Set to 0.

CONSTRAINTS
The second MBLK (linked by b_cont) will contain space to hold the statistics of size Ext_mib_t.

The second MBLK will be empty (b_rptr == b_wptr). After writing the information (statistics information) to the second MBLK, the drivers must move the b_wptr to the size that was written to the second MBLK (in this case by the size of the Ext_mib_t). The drivers can cast the b_rptr to be Ext_mib_t *, if it has to change some fields.

Drivers must not modify the b_rptr.

Even if driver supports only a subset of the statistics groups in the Ext_mib_t structure, it must copy the complete structure.

Ext_mib_t is defined in <sys/mib.h>

SEE ALSO

driver_controlp(9F)
DL_HP_GET_64BIT_STATS_REQ(9G)

NAME

DL_HP_GET_64BIT_STATS_REQ - Request to get 64-bit statistics from an interface.

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

This primitive is sent to drivers using the driver_controlp routine to get the 64-bit statistics. The associated structure for this request is dl_hp_get_64bit_stats_req_t.

STRUCTURE MEMBERS

typedef struct {
    uint32_t        dl_primitive;
    uint32_t        dl_reserved1;
    uint32_t        dl_reserved2;
} dl_hp_get_64bit_stats_req_t;

dl_primitive Set to DL_HP_GET_64BIT_STATS_REQ

dl_reserved1 Reserved for future use. Set to 0.

dl_reserved2 Reserved for future use. Set to 0.

CONSTRAINTS

The second MBLK (linked by b_cont) will contain the space to hold the statistics of size Ext_64bit_mib_t.

Ext_64bit_mib_t is defined in <sys/mib.h>.

The second MBLK will be empty (b_rptr == b_wptr). After writing the information (include both 64-bit and standard 32-bit statistics) to the second MBLK, the drivers must move the b_wptr to the size that was written to the second MBLK. (In this case, by the size of the Ext_64bit_mib_t).

Drivers must not modify the b_rptr.

If the driver does not have any interface specific statistics, the b_wptr must still be set to the end of Ext_64bit_mib_t, and zero the interface-specific part of the data structure.

The driver is responsible for following RFC 2863 and other RFCs, which define the interface specific MIBs, to fill out all the fields.

For drivers that support 64-bit statistics, all 32-bit statistics requests will be converted to DL_HP_GET_64BIT_STATS_REQ if the feature bit DL_HP_DRV_64BIT_MIB is set when the request is received. If a driver does not change this bit, no 32-bit request will be received. However, if a driver dynamically resets this feature bit, it will receive both 32-bit and 64-bit statistics requests depending on whether the features bit is set at the point the request is received by DLPI.

SEE ALSO

driver_controlp(9F)
NAME
dl_hp_getinfo_t - Generic data type to get information from HP-DLPI.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
The data type that must be passed in the valuep parameter to dlpi_propp for the
DLPI_PROP_MCAST_INFOP and DLPI_PROP_INSTANCE_INFOP properties.
Except for the dhg_modifier field, all other fields are set by dlpi_propp on a successful
return.
The data type for each item in the definition depends on the property being requested. The
dhg_datap being returned could be an MBLK or a memory block (see the property definitions
in dl_hp_prop_t(9F) for the type of dhg_datap returned). The caller must not modify the contents
of the MBLK or data block in the dhg_datap in any way and must invoke the function provided
(dhg_free_funcp).
The caller must always use dhg_len to walk through the list of items in the buffer (the caller
must not use sizeof to determine the size of each item.)

STRUCTURE MEMBERS
typedef struct dl_hp_getinfo {
    dl_hp_search_info_t  *dhg_modifier;
    uint32_t              dhg_count;
    uint32_t              dhg_len;
    void                  (*dhg_free_funcp)(uint8_t *datap);
    uint8_t              *dhg_datap;
    void                  *dhg_rsvdp;
} dl_hp_getinfo_t;

dhg_modifier  Caller must set this to NULL unless the property allows specification of
some criteria to limit the information requested. For example, the
DLPI_PROP_INSTANCE_INFOP property allows the caller to request
information about driver instances that match a specific criteria. In this
case, this field must point to structure (for example,
dl_hp_search_info_t) specifying the search criteria.
When calling DLPI_PROP_MCAST_INFOP, this element must be set to
NULL.

dhg_count    HP-DLPI will set this field. Specifies the total number of items being
returned.

dhg_len     HP-DLPI will set this field. Specifies the length of each item.

dhg_free_funcp  HP-DLPI will set this field.
Specifies the callback function that must be called after the driver is done
processing the requested information passed in the dhg_datap field.
This function must be passed the pointer passed in dhg_datap.
The caller must not modify dhg_datap.

dhg_datap  If dhg_count is set to 0, HP-DLPI was unable to find a match based on
the search criteria specified by the caller. In which case, dhg_datap will
be set to NULL. If HP-DLPI is able to find a match, it contains a pointer
to a memory block of size \textit{dhg\_len} times \textit{dhg\_count}. This is the information returned to the caller.

Set to NULL if no information was returned (that is, \textit{dhg\_count} was 0).

If information was returned, each item is of length \textit{dhg\_len}. The \textit{dhg\_len} must be used to walk through the list of items in the data passed.

The data type of the item returned depends on the property requested and is described in the definition of the property. After reading the data passed, the caller must call the \textit{dhg\_free\_funcp} function pointer with \textit{dhg\_datap} as the parameter. See \textit{dl\_hp\_prop\_t}(9F) for more information.

\textit{dhg\_rsvdp} \hspace{1cm} Reserved for future use. Must be set to 0.

**SEE ALSO**

\textit{dlpi\_propp}(9F), \textit{dl\_hp\_prop\_t}(9S), \textit{dl\_hp\_search\_info\_t}(9S), \textit{sizeof}(9F)
NAME
dl_hp_hdr_length_t - Data type for OOP type DL_HP_OOP_HDR_LENGTH.

SYNOPSIS
#include <sys/dlpi_ext.h>

DESCRIPTION
Contains the length of the LLC header of the packet. This OOP type is defined in
dl_hp_oop_type_t.

STRUCTURE MEMBERS

typedef struct dl_hp_hdr_length {
    uint16_t       dhhl_reserved;
    uint16_t       dhhl_length;
} dl_hp_hdr_length_t;

dhhl_reserved       Reserved element.
dhhl_length          LLC plus OOP header length.

SEE ALSO
dl_hp_oop_type_t(9S)
NAME
DL_HP_HW_RESET_REQ - Request to reset physical interface.

SYNOPSIS
#include <sys/dlpi_ext.h>

DESCRIPTION
This primitive is sent to drivers using the driver_controlp routine to reset the physical interface. The associated structure for this request is dl_hp_hw_reset_req_t structure.

The DL_HP_HW_RESET_REQ primitive described in this manpage may be different from the definition in DLPI Programmer's Guide. The information provided here is applicable only to tightly-coupled drivers.

STRUCTURE MEMBERS
typedef struct {
    uint32_t dl_primitive;
} dl_hp_hw_reset_req_t;

dl_primitive Set to DL_HP_HW_RESET_REQ

CONSTRAINTS
Drivers need to generate link events (link DOWN and UP) when there is a change in any of the driver properties (in this case the operating state may change). See the HP-UX 11i v2 Driver Development Guide for information on event generation.

SEE ALSO
driver_controlp(9F)
NAME

dl_hp_hw_state_t - Hardware state of the interface.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

Hardware state of the interface. This enumeration is used to set hardware state using the
DLPI_PROP_HDW_STATE property.

STRUCTURE MEMBERS

typedef enum dl_hp_hw_state {
   DL_HP_HW_UP,
   DL_HP_HW_DOWN
} dl_hp_hw_state_t;

DL_HP_HW_UP Hardware state is UP.
DL_HP_HW_DOWN Hardware state is DOWN.

SEE ALSO

dlpi_propp(9F)
NAME

dl_hp_ifadmin_state_t - Administration state of the interface (ifAdminStatus).

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

The desired state of the interface. This definition is used to set ifAdmin state of the interface using the DLPI_PROP_IFADMIN property.

STRUCTURE MEMBERS

typedef enum dl_hp_ifadmin_state {
    DL_HP_IFADMIN_UP,
    DL_HP_IFADMIN_DOWN,
    DL_HP_IFADMIN_TEST
} dl_hp_ifadmin_state_t;

DL_HP_IFADMIN_UP Administration status of the interface is UP.
DL_HP_IFADMIN_DOWN Administration status of the interface is DOWN.
DL_HP_IFADMIN_TEST Not Supported.

SEE ALSO

dlpi_propp(9F)
dl_hp_instance_info_t(9S)

NAME
dl_hp_instance_info_t - Structure returned on use of DL_PROP_INTERFACE_INFOP HP-DLPI property.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
This structure is returned in response to the use of the DL_PROP_INTERFACE_INFOP HP-DLPI property. This structure contains the interface information returned by a DLS provider.

STRUCTURE MEMBERS
typedef struct dl_hp_instance_info {
    uint32_t                dii_version;
    uint32_t                dii_instance_num;
    dl_hp_mac_type_t        dii_mac_type;
    uint32_t                dii_mac_addr_len;
    uint32_t                dii_nm_id;
    dl_hp_encaps_type_t     dii_encaps;
    uint32_t                dii_mtu;
    uint32_t                dii_major_num;
    dl_hp_drv_features_one_t dii_features_one;
    uint32_t                dii_features_two;
    uint32_t                dii_features_three;
    uint8_t                 dii_mac_addr[DL_HP_MAX_MAC_ADDR_LEN];
    char                    dii_hdw_path[DL_HP_MAX_HDW_PATH_LEN];
    char                    dii_drv_name[DL_HP_MAX_DRIVER_NAME_LEN];
    char                    dii_arpmod_name[DL_HP_MAX_ARPMOD_NAME_LEN];
    uint32_t                dii_max_mcast;
    uint32_t                dii_max_vlan;
    dl_hp_ifadmin_state_t   dii_ifadmin;
    dl_hp_link_state_t      dii_ifoper;
    dl_hp_duplex_mode_t     dii_duplex;
    uint32_t                dii_speed;
    uint32_t                (*dii_outputp)(void *drv_output_hdlp,
                                           mblk_t *mblkp,
                                           dl_hp_pkt_type_t pkt_type,
                                           void *rsvdp);
    uint32_t                (*dii_build_hdrp)(void *drv_output_hdlp,
                                            dl_hp_src_llc_info_t *src_infop,
                                            dl_hp_dest_llc_info_t *dset_infop,
                                            dl_hp_llc_info_t *hdr_infop,
                                            void *rsvd1p,
                                            void *rsvd2p);
} dl_hp_instance_info_t;
void                        *dii_output_hdlp;
void                        *dii_dlipi_hdlp;
uint32_t                    dii_ppa;
dii_next_offset;
uint64_t                    dii_reserved[2];

} dl_hp_instance_info_t;

dii_version
The lowest HP-DLPI version with which this interface is compatible.
dii_instance_num
Maps one-to-one with a physical interface belonging to a particular
class.
dii_mac_type
This is a MAC type from one of the supported MAC types in the
dl_hp_mac_type_t structure.
dii_mac_addr_len
The length, in bytes, of the MAC address.
dii_nm_id
The Network Management ID. Each interface has a unique NMID.
dii_encaps
Specifies the encapsulations supported by the drivers. For supported
encapsulations, see dl_hp_encaps_type_t(9S).
dii_mtu
The current MTU of the link in bytes.
dii_major_num
Identifies DLPI's major number. For loosely-coupled drivers, the
major number is different from HP-DLPI's major number.
dii_features_one
Identifies the list of supported features by the driver instance. For
a list of supported features, see dl_hp_drv_features_one_t(9S).
dii_features_two
Reserved for future use. It must be set to 0.
dii_features_three
Reserved for future use. It must be set to 0.
dii_mac_addr
The MAC address of the card. The size of MAC address must be
equal to dii_mac_addr_len.
The MAC address is in canonical format for Ethernet and FDDI and
in wire format for Token Ring.
dii_hdw_path
The hardware path of the physical interface as a (null terminated)
character string. The length of the hardware path could be a
maximum of DL_HP_MAX_HDW_PATH_LEN bytes, including NULL
termination. DL_HP_MAX_HDW_PATH_LEN is currently defined as
100.
dii_drv_name
The driver name as a (null terminated) character string. The name
must be less than or equal to DL_HP_MAX_DRIVER_NAME_LEN
characters and must be NULL terminated.
DL_HP_MAX_DRIVER_NAME_LEN is currently defined as 64.
dii_arpmod_name
A character string that identifies the ARP (STREAMS) module name
for this driver, if one is present. This field is currently used by Token
Ring drivers.
The name must be less than or equal to
DL_HP_MAX_ARPMOD_NAME_LEN characters and must be NULL
terminated. DL_HP_MAX_ARPMOD_NAME_LEN is currently defined
as 64.
dii_max_mcast
The driver specified maximum number of unique multicast
addresses supported by the driver or NIC in the range
1-0x7FFFFFFF.
dii_max_vlan
Not supported in HP-UX 11i v3.
dii_ifadmin
Identifies the administration status (ifAdmin) of the interface. For
a list of supported ifAdmin values, see dl_hp_ifadmin_state_t(9S).
**dii_ifoper**
Identifies the operational status (ifOper) of the interface. For a list of supported ifOper values, see `dl_hp_link_state_t(9S)`.

**dii_duplex**
Specifies the duplicity setting of the driver instance. For supported values of duplicity, see `dl_hp_duplex_t(9S)`.

**dii_speed**
Specifies the speed of the interface in Mb/s.

**dii_outputp**
The driver outbound routine. See `driver_outputp(9F)` for information.

**dii_build_hdrp**
The function that identifies the driver routine to build the MAC/LLC header in the outbound path. See `driver_build_hdrp(9F)` for information.

**dii_output_hdlp**
The driver-specified opaque (to HP-DLPI) handle that is passed to the `dii_outputp` and `dii_build_hdrp` outbound routines.

**dii_dlpi_hdlp**
HP-DLPI's per-interface handle. This is the opaque handle that drivers must use in calls to HP-DLPI interfaces.

**dii_ppa**
Specifies the physical point of attachment (PPA) of the interface.

**dii_next_offset**
Specifies the offset from the beginning of the returned memory block, to the next `dl_hp_instance_info_t` structure.

**dii_reserved**
Reserved for future use. It must be set to 0.

**SEE ALSO**
`dlpi_propp(9F), dl_hp_mac_type_t(9S), dl_hp_encaps_type_t(9S), dl_hp_features_one_t(9S), dl_hp_ifadmin_state_t(9S), dl_hp_link_state_t(9S), dl_hp_duplex_t(9S), driver_outputp(9F), driver_build_hdrp(9F), dl_hp_instance_info_t(9S)`
NAME

dl_hp_llc_info_t - LLC header information.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

This structure contains the LLC header information. This structure is returned from driver's build header routine.

STRUCTURE MEMBERS

typedef struct dl_hp_llc_info {
    dl_hp_pkt_type_t          dli_pkt_type;
    uint32_t                  dli_llc_length;
    mblk_t                   *dli_mblkp;
    uint64_t                  dli_reserved1[2];
    uint64_t                  dli_reserved2[2];
} dl_hp_llc_info_t;

- **dli_pkt_type** Identifies the packet type. See *dl_hp_pkt_type_t(9S)* for a list of supported packet types.
- **dli_llc_length** Identifies the length of LLC header, in bytes. This does not include the size of OOP data, if any.
- **dli_mblkp** A pointer to a MBLK that contains LLC header and space for OOP data, if any.
- **dli_reserved1** Reserved for future use. It must be set to 0.
- **dli_reserved2** Reserved for future use. It must be set to 0.

SEE ALSO

dl_hp_pkt_type_t(9S)
NAME
dl_hp_mac_type_t - MAC Type definitions.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
MAC Type of the interface. The dhc_mac_type element of dl_hp_create_info_t is of this type. The dl_hp_create_info_t structure is the associated structure for the DL_HP_OP_CREATE operation in dlpi_propp.

STRUCTURE MEMBERS
typedef enum dl_hp_mac_type {
    DL_HP_DEV_8023,
    DL_HP_DEV_8024,
    DL_HP_DEV_8026,
    DL_HP_DEVETHER,
    DL_HP_DEV_HDLC,
    DL_HP_DEV_CHAR,
    DL_HP_DEV_CTCA,
    DL_HP_DEV_OTHER,
    DL_HP_DEV_FC,
    DL_HP_DEV_ATM,
    DL_HP_DEV_IPATM,
    DL_HP_DEV_X25,
    DL_HP_DEV_ISDN,
    DL_HP_DEV_HIPPI,
    DL_HP_DEV_100VG,
    DL_HP_DEV_100VGTPR,
    DL_HP_DEV_ETH_CSMA,
    DL_HP_DEV_100BT,
    DL_HP_DEV_IB
} dl_hp_mac_type_t;

HP-DLPI supports the following MAC types for tightly-coupled drivers:
DL_HP_DEV_8023 — IEEE 8023
DL_HP_DEV_ETHER — Ethernet Interface (802.3 encapsulation is assumed to be supported if the driver sets its MAC type to this value).

Loosely-coupled drivers can be any of the types defined in the structure.

SEE ALSO
dl_hp_create_info_t(9S), dlpi_propp(9F)
dl_hp_oop_hdr_t(9S)

NAME
dl_hp_oop_hdr_t - Structure defining each OOP's header.

SYNOPSIS
#include <sys/dlpi_ext.h>

DESCRIPTION
OOP header. Structure definition for the OOP type header containing the type and size values.

STRUCTURE MEMBERS

typedef struct dl_hp_oop_hdr {
    uint16_t    dhoh_type;
    uint16_t    dhoh_size;
} dl_hp_oop_hdr_t;

dhoh_type    OOP type. See dl_hp_oop_type_t(9S) for more information.
dhoh_size    Size of the OOP data.

SEE ALSO

dl_hp_oop_type_t(9S)
NAME

dl_hp_oop_template_t - Structure to be passed when using the DLPI_PROP_OOP_INFO property.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

The structure containing inbound OOP data template and the associated information. The OOP template passed in this structure must be used by the driver for every packet that is passed upstream to HP-DLPI.

The dhot_mblkp returned by HP-DLPI must never be freed. The dhot_mblkp present in subsequent calls to DLPI_PROP_OOP_INFO will overwrite the previously returned dhot_mblkp.

For more information on OOP, see the HP-UX 11i v2 Driver Development Guide.

STRUCTURE MEMBERS

typedef struct dl_hp_oop_template {
    mblk_t            *dhot_mblkp;
    uint32_t          dhot_drv_pad;
    uint32_t          dhot_oop_size;
    uint32_t          dhot_oop_copy_size;
    void              *rsvdp;
    uint32_t          reserved[2];
} dl_hp_oop_template_t;

dhot_mblkp          HP-DLPI sets dhot_mblkp pointer to the MBLK that contains the OOP data template. Valid only if dhot_oop_size is valid. Drivers must not free this MBLK at any time.

dhot_drv_pad         Drivers will set this element to specify the padding in the inbound packet required to avoid IP pull-ups. See HP-UX 11i v2 Driver Development Guide for more information.

dhot_oop_size        HP-DLPI will pass OOP data size in bytes.

dhot_oop_copy_size   The end-of-options OOP may contain padding and other non-data bytes. Because the drivers do not need to copy them and because they might make dhot_oop_size a non-multiple of 8 bytes, the copyable portion of the OOP template is specified in dhot_oop_copy_size. HP-DLPI guarantees that the copyable size will always be a multiple of 8.

rsvdp                Reserved for future use. Must be set to NULL.

reserved             Reserved for future use. Must be set to 0.

SEE ALSO

dlpi_propp(9F)
NAME

dl_hp_oop_type_t - Out-of-Packet data types.

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

These enumerated types are OOP data types that are passed between HP-DLPI and the driver. In the inbound and outbound path, the actual payload of the packet is preceded by the OOP data based on the feature bits set by the driver in dhc_features_one element of dl_hp_create_info_t.

The OOP data communicated between HP-DLPI and the driver is different from the OOP data communicated between HP-DLPI and the upper layer. See the HP-UX 11i v2 Driver Development Guide for more information OOP mechanism between HP-DLPI and the driver.

Each OOP is a combination of Type-Size-Data. Each data type has a structure associated with it. OOP itself has a header of its own containing the Type-Size of OOP data. The structure is dl_hp_oop_hdr_t. See each OOP type for associated structures.

STRUCTURE MEMBERS

typedef enum dl_hp_oop_type {
    DL_HP_OOP_PAD = 0x0001,
    DL_HP_OOP_CKO = 0x0002,
    DL_HP_OOP_VLAN_TAG = 0x0003,
    DL_HP_OOP_TOS = 0x0004,
    DL_HP_OOP_HDR_LENGTH = 0x0005,
    DL_HP_OOP_TCPSEG_IPV4 = 0x0006,
    DL_HP_OOP_TCPSEG_IPV6 = 0x0007,
    DL_HP_OOP_PORT = 0x0008,
    DL_HP_OOP_EOO = 0x00ff,
    DL_HP_TYPE_DATA_INVALID = 0x1000,
    DL_HP_TYPE_MASK = 0xefff
} dl_hp_oop_type_t;

DL_HP_OOP_PAD: Padding type.
DL_HP_OOP_CKO: Reserved for future use.
DL_HP_OOP_VLAN_TAG: Reserved for future use.
DL_HP_OOP_TOS: Reserved for internal use.
DL_HP_OOP_HDR_LENGTH: Length of the header (OOP + LLC header) in a packet. Associated information for this OOP type is dl_hp_hdr_length_t.
DL_HP_OOP_EOO: End of Options. Delimiter which separates the OOP and the payload.
DL_HP_TYPE_DATA_INVALID: If this bit is set in any of the options types, the data associated with the type is be invalid.
DL_HP_TYPE_MASK: Mask to get the actual OOP type values.

SEE ALSO

dl_hp_hdr_length_t(9S), dl_hp_oop_hdr_t(9S), dl_hp_pri_info_t(9S), dl_hp_vlan_tag_t(9S)
**NAME**

dl_hp_op_param_type_t - Operating parameters that drivers need to set.

**SYNOPSIS**

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

**DESCRIPTION**

The enumerated type is used to specify which of the driver's operating parameters are being set. These are used in the dl_hp_op_param_t structure. This enumeration contains bit wise flags to allow setting of multiple operating parameters for a driver instance.

**STRUCTURE MEMBERS**

```c
typedef enum dl_hp_op_param_type {
    DL_HP_SPEED_SET,
    DL_HP_DUPELEX_SET,
    DL_HP_MTU_SET,
    DL_HP_AUTO_NEG_SENSE_SET
} dl_hp_op_param_type_t;
```

**dl_hp_op_param_type_t Members**

- **DL_HP_SPEED_SET**
  Sets the speed of the interface. The `dop_speed` element of the `dl_hp_op_param_t` structure contains the valid speed value.

- **DL_HP_MTU_SET**
  Sets the MTU of the interface. The `dop_mtu` element of the `dl_hp_op_param_t` structure contains the valid MTU value.

- **DL_HP_DUPELEX_SET**
  Sets the duplex type of the interface. The `dop_duplex` element of the `dl_hp_op_param_t` structure contains the valid duplex setting.

- **DL_HP_AUTO_NEG_SENSE_SET**
  Sets the Auto Negotiation/Auto Sense value of the interface. The `dop_auto_neg_sense` element of the `dl_hp_op_param_t` structure contains valid Auto Negotiation/Auto Sense value.

**SEE ALSO**

`dlpi_propp(9F), dl_hp_op_param_t(9S)}`
dl_hp_op_t(9S)

NAME

dl_hp_op_t - Operation codes supported for dlpi_propp routine.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

Operators for the HP-DLPI dlpi_propp property exchange interface. This corresponds to the opcode argument in dlpi_propp.

STRUCTURE MEMBERS

typedef enum dl_hp_op {
    DL_HP_OP_GET    = 1,
    DL_HP_OP_SET    = 2,
    DL_HP_OP_CREATE = 3,
    DL_HP_OP_DELETE = 4,
    DL_HP_OP_TEST_Set = 5,
    DL_HP_OP_SEARCH = 6
} dl_hp_op_t

DL_HP_OP_GET Get information from HP-DLPI.
DL_HP_OP_SET Set a property in HP-DLPI.
DL_HP_OP_CREATE Register with HP-DLPI.
DL_HP_OP_DELETE Delete a driver instance.
DL_HP_OP_TEST_Set Not supported. Reserved for future use.
DL_HP_OP_SEARCH Not supported. Reserved for future use.

SEE ALSO

dlpi_propp(9F)
NAME
dl_hp_pkt_type_t - Packet types supported.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
This enumeration defines all packet types. HP-DLPI supports only a subset of the packet types
defined here.

STRUCTURE MEMBERS
typedef enum dl_hp_pkt_type {
    DL_HP_ETHER_PKT          = 0,
    DL_HP_IEEE8023XSAP_PKT   = 1,
    DL_HP_SNAP8023_PKT       = 2,
    DL_HP_ETHERXT_PKT        = 3,
    DL_HP_SNAP8024_PKT       = 4,
    DL_HP_IEEE8023_PKT       = 5,
    DL_HP_SNAP8023XT_PKT     = 6,
    DL_HP_SNAP8024XT_PKT     = 7,
    DL_HP_SNAPFDDI_PKT       = 8,
    DL_HP_SNAP8025_PKT       = 9,
    DL_HP_IEEE8025_PKT       = 10,
    DL_HP_ATR_PKT            = 11,
    DL_HP_FDDI_UI_PKT        = 12,
    DL_HP_SNAPFDDI_LLA_PKT   = 13,
    DL_HP_FDDI_LLA_PKT       = 14,
    DL_HP_FCSNAP_PKT         = 15,
    DL_HP_FC8022_PKT         = 16,
    DL_HP_IEEE8025XSAP_PKT   = 17,
    DL_HP_FDDIXSAP_PKT       = 18,
    DL_HP_FDDI_PKT           = 253,
    DL_HP_UNKNOWN_PKT        = 254,
    DL_HP_RAW_PKT            = 255
} dl_hp_pkt_type_t;

HP-DLPI supports the following packet types:

**DL_HP_ETHER_PKT**   Ethernet packet type.

```
<table>
<thead>
<tr>
<th>DA</th>
<th>SA</th>
<th>TYPE</th>
<th>PAYLOAD</th>
</tr>
</thead>
</table>
```

6 bytes   6 bytes   2 bytes

**DL_HP_IEEE8023XSAP_PKT**   IEEE 802.3 Extended SAP packet type.

```
<table>
<thead>
<tr>
<th>L</th>
<th>D</th>
<th>S</th>
<th>C</th>
<th>DS</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>S</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>G</td>
<td>A</td>
<td>A</td>
<td>R</td>
<td>HDR_FILL</td>
<td>S</td>
</tr>
<tr>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>
```
### DL_HP_SNAP8023_PKT

SNAP 802.3 packet type.

<table>
<thead>
<tr>
<th>L</th>
<th>E</th>
<th>D</th>
<th>S</th>
<th>C</th>
<th>P</th>
<th>A</th>
<th>Y</th>
<th>L</th>
<th>O</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>A</td>
<td>A</td>
<td>R</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
</tr>
</tbody>
</table>

### DL_HP_IEEE8023_PKT

IEEE 802.3 packet type.

<table>
<thead>
<tr>
<th>L</th>
<th>E</th>
<th>D</th>
<th>S</th>
<th>C</th>
<th>P</th>
<th>A</th>
<th>Y</th>
<th>L</th>
<th>O</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>A</td>
<td>A</td>
<td>R</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
</tr>
</tbody>
</table>

### DL_HP_SNAPFDDI_PKT

FDDI SNAP packet type.

<table>
<thead>
<tr>
<th>L</th>
<th>E</th>
<th>D</th>
<th>S</th>
<th>C</th>
<th>P</th>
<th>A</th>
<th>Y</th>
<th>L</th>
<th>O</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>A</td>
<td>A</td>
<td>R</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
</tr>
</tbody>
</table>

### DL_HP_SNAP8025_PKT

Token Ring SNAP packet type.

<table>
<thead>
<tr>
<th>L</th>
<th>E</th>
<th>D</th>
<th>S</th>
<th>C</th>
<th>P</th>
<th>A</th>
<th>Y</th>
<th>L</th>
<th>O</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>A</td>
<td>A</td>
<td>R</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
<td>T</td>
<td>P</td>
<td>P</td>
<td>L</td>
</tr>
</tbody>
</table>

### DL_HP_IEEE8025_PKT

IEEE 802.5 packet type.
DL_HP_FDDI_PKT  FDDI SAP packet type.

DL_HP_IEEE8025XSAP_PKT  Token Ring Extended SAP packet type.

DL_HP_RAW_PKT  Raw HP-DLPI packet. Drivers must not modify the packet in any form.

SEE ALSO
dlpi_propp(9F)
NAME

dl_hp_promisc_state_t - Promiscuous mode information that is passed for the
DLPI_PROP_PROMISC_INFO property.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

This enumerated type is used during DLPI_PROP_PROMISC_INFO property call to get the
current promiscuous mode information for an interface. This is a bit wise enumeration that will
return all the promiscuous levels currently enabled.

STRUCTURE MEMBERS

typedef enum dl_hp_promisc_state {
    DL_HP_PHYSICAL_LEVEL   = 0x1,
    DL_HP_SAP_LEVEL        = 0x2,
    DL_HP_MULTI_LEVEL      = 0x4
} dl_hp_promisc_state_t;

DL_HP_PHYSICAL_LEVEL     Physical level promiscuous mode is enabled.
DL_HP_SAP_LEVEL         SAP level promiscuous mode is enabled.
DL_HP_MULTI_LEVEL       Multicast level promiscuous mode enabled.

SEE ALSO

dlpi_prop(9F)
NAME

dl_hp_prop_t - List of HP-DLPI properties for a driver.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

This enumeration type contains the properties that are handled by HP-DLPI. The properties
listed here are for tightly-coupled and loosely-coupled drivers. The properties are passed as part
of the dlpi_propp call to HP-DLPI. Each property is explained along with the values HP-DLPI
expects for dlpi_hdlp, valuep, and opcode when calling dlpi_propp.

STRUCTURE MEMBERS

typedef enum dl_hp_prop {
    DLPI_PROP_VERSION,
    DLPI_PROP_DRV_INSTANCEP,
    DLPI_PROP_WAKEUPP,
    DLPI_PROP_EVENTP,
    DLPI_PROP_HDW_STATE,
    DLPI_PROP_MAC_ADDR,
    DLPI_PROP_RESERVED1,
    DLPI_PROP_INBOUNDP,
    DLPI_PROP_NMID,
    DLPI_PROP_FEATURES_ONE,
    DLPI_PROP_IFADMIN,
    DLPI_PROP_MCAST_INFOP,
    DLPI_PROP_PROMISC_INFO,
    DLPI_PROP_PROTOCOL_INFOP,
    DLPI_PROP_VLAN_INFOP,
    DLPI_PROP_OOP_INFOP,
    DLPI_PROP_INSTANCE_INFOP,
    DLPI_PROP_MODULEP,
    DLPI_PROP_MOD_ACQUIRE_PORT,
    DLPI_PROP_MOD_RESERVED,
    DLPI_PROP_MOD_RELEASE_PORT,
    DLPI_PROP_MOD_ASSOCIATE_PORT,
    DLPI_PROP_MOD_DISASSOCIATE_PORT,
    DLPI_PROP_MOD_UPDATE_INSTANCEP,
    DLPI_PROP_MOD_CONTROLP,
    DLPI_PROP_PPA_RANGE
    DLPI_PROP_RESERVED2,
    DLPI_PROP_OP_PARAM,
    DLPI_PROP_OOP_OUT_INFOP,
    DLPI_PROP_RESERVED3,
    DLPI_PROP_TRAP_OPTZER,
    DLPI_PROP_IFLASTCHANGE,
    DLPI_PROP_DRIVER_INFOP,
    DLPI_PROP_MODULE_INFOP,
} dl_hp_prop_t;
**DLPI_PROP_VERSION**

*dlpi_hdlp*: Can be set to NULL, if a HP-DLPI provided handle is not available.

*opcode*: DL_HP_OP_GET

*valuep*:

**In**: Pointer to uint32_t

**Out**: The pointer content is set to the HP-DLPI version number.

This property is valid for tightly coupled and loosely coupled drivers. HP-DLPI recommends the drivers to get this property as the first step before registering its instance with HP-DLPI. Getting this property will help drivers know what version of DLPI they are working with and to see if the drivers and DLPI are compatible.

After getting this property, the driver must set the value received in *dhc_version* element of the *dl_hp_create_info_t* structure during create operation. The *dhc_version* field is the version of HP-DLPI with which the driver is compatible.

The HP-DLPI version number will be set to 2 for the HP-UX 11i v2.

HP-DLPI Version 2 will not be compatible with future HP-DLPI versions.

**DLPI_PROP_DRV_INSTANCEP**

*dlpi_hdlp*: Must be set to NULL.

*opcode*: DL_HP_OP_CREATE

*valuep*:

**In**: Pointer to *dl_hp_create_info_t*.

**Out**: Pointer to the same structure with *dhc_dlpi_hdlp* field of *dl_hp_create_info_t* filled with pointer to the HP-DLPI private handle.

**Description**:

This property is valid for tightly coupled and loosely coupled drivers. This property is provided to accomplish the Create operation of a driver instance. In addition, it allows drivers to register some of their properties with the HP-DLPI.

**DLPI_PROP_WAKEUPP**

*dlpi_hdlp*: May be set to NULL, if a HP-DLPI provided handle is not available

*opcode*: DL_HP_OP_GET

*valuep*:

**In**: Pointer to HP-DLPI *wakeup* routine. See *dlpi_wakeup(9F)* for more information.

**Out**: Filled with a function pointer.

**Description**:

This property is valid only for tightly coupled drivers.

This property is used to get the pointer to the HP-DLPI function. The HP-DLPI *wakeup* function must be called on completion of all control requests.
All control requests passed via the `driver_controlp` interface are assumed to be asynchronous in nature and require the HP-DLPI `wakeup` function to be called on completion of the request.

**DLPI_PROP_EVENTP**

- `dlpi_hdlp`: Can be set to NULL, if a HP-DLPI provided handle is not available
- `opcode`: DL_HP_OP_GET
- `valuep`:
  - **In**: Pointer to HP-DLPI event handler routine. See `dlpi_eventp(9F)` for more information.
  - **Out**: Filled with a function pointer.

This property is valid only for tightly coupled drivers.

This property is used to get the pointer to the HP-DLPI event handler. The HP-DLPI event handler function can be used to send HP-DLPI defined events triggered by drivers.

**DLPI_PROP_HDW_STATE**

- `dlpi_hdlp`: Must be the HP-DLPI provided handle.
- `opcode`: DL_HP_OP_SET
- `valuep`:
  - **In**: Pointer to a variable of type `dl_hp_hw_state_t`.

This property is valid for tightly coupled and loosely coupled drivers. Drivers must call `dlpi_propp` with this property to update changes in the hardware state of the interface.

Driver must generate link DOWN and UP events if the hardware state of the driver instance changes. See the `HP-UX 11i v2 Driver Development Guide` for more information.

Tightly coupled drivers must call this property after the initialization completes. Until such a time, the driver instance will not be visible to normal users. Drivers must also update HP-DLPI with correct operating parameters (for example, MAC address and MTU) before making this property call.

**DLPI_PROP_MAC_ADDR**

- `dlpi_hdlp`: Must be the HP-DLPI provided handle.
- `opcode`: DL_HP_OP_SET
- `valuep`:
  - **In**: Pointer to `uint8_t`.

This property is valid for tightly coupled and loosely coupled drivers.

This property is used to communicate changes in the MAC address of the interface to HP-DLPI. This property must be called by drivers whenever the interface MAC address changes.

The MAC address must be in canonical format for Ethernet and FDDI and in wire format for Token Ring. The length of the MAC address must be same as the value specified.
by the driver during `DL_HP_OP_CREATE` in the
dhc_mac_addr_len field of the `dl_hp_create_info_t`
structure.

Tightly coupled drivers must generate events if the MAC
address of the driver instance changes. See the *HP-UX 11i
v2 Driver Development Guide* for more information.

### DLPI_PROP_RESERVED1

Reserved by HP-DLPI.

### DLPI_PROP_INBOUND

dlp_hdlp: Must be the HP-DLPI provided handle.

*opcode*: `DL_HP_OP_GET`

*valuep*:

**In**: Pointer to the HP-DLPI inbound processing routine. See `dlpi_inboundp(9F)` for more information.

**Out**: Filled with a function pointer.

This property is valid only for tightly coupled drivers.

Tightly coupled drivers can invoke this property, any time
after successful registration of a driver instance. This
property is used to get the pointer to the HP-DLPI inbound
routine, which has to be invoked for frames coming in on
that interface.

### DLPI_PROP_NMID

dlp_hdlp: Must be the HP-DLPI provided handle.

*opcode*: `DL_HP_OP_GET` / `DL_HP_OP_SET`

*valuep*:

For `DL_HP_OP_GET`:

**In**: Pointer to `uint32_t`.

**Out**: Filled with NMID.

For `DL_HP_OP_SET`:

**In**: Pointer to `uint32_t`, filled with NMID to be set for
the interface.

**Get/Set** the NM ID of the interface.

This property is valid for tightly coupled and loosely
coupled drivers. However, only loosely coupled drivers
are permitted to modify NMID.

For tightly coupled drivers, the NMID is allocated by
HP-DLPI, and tightly coupled drivers can get it from
HP-DLPI using this property.

### DLPI_PROP_FEATURES_ONE

dlp_hdlp: Must be the HP-DLPI provided handle.

*opcode*: `DL_HP_OP_SET`

*valuep*:

**In**: Pointer to `dl_hp_drv_features_one_t`.

This property is valid for tightly coupled drivers.

This property allows tightly coupled drivers to set or
change the features supported by drivers.

Driver must generate events if the features of the driver
instance changes. See the *HP-UX 11i v2 Driver Development
Guide* for more information.
DLPI_PROP_IFADMIN  
**opcode:** DL_HP_OP_SET  
**valuep:**  
**In:** Pointer to dl_hp_ifadmin_state_t.  
This property is valid for tightly coupled drivers.  
Tightly coupled drivers must call this property to set or change the ifAdmin value of the interface.  
Driver must generate events if the ifAdmin state of the driver instance changes. See the HP-UX 11i v2 Driver Development Guide for more information.

DLPI_PROP_MCAST_INFO  
**dlpi_hdlp:** Must be the HP-DLPI provided handle.  
**opcode:** DL_HP_OP_GET  
**valuep:**  
**In:** Pointer to dl_hp_getinfo_t.  
**Out:** Fills up the dl_hp_getinfo_t structure and sets the dhg_datap element to point to an MBLK containing the list of unique multicast addresses enabled on the interface.  
The MBLK->b_rptr will point to the first multicast address in the list of multicast address returned.  
This property is valid for tightly coupled drivers. This property can be used to get the list of unique multicast addresses enabled on an instance and the multicast addresses themselves.  
HP-DLPI will allocate the dhg_datap element of dl_hp_get_info_t structure.  
The dhg_datap element of the dl_hp_getinfo_t structure must be freed using the dhg_free_funcp routine provided in the same structure. No locks must be held across the call to dhg_free_funcp.

DLPI_PROP_PROMISC_INFO  
**dlpi_hdlp:** Must be the HP-DLPI provided handle.  
**opcode:** DL_HP_OP_GET  
**valuep:**  
**In:** Pointer to dl_hp_promisc_state_t.  
**Out:** Filled with promiscuous level. HP-DLPI will return a bitmask containing all the levels that are enabled.  
This property is valid for tightly coupled drivers.  
Get the promiscuous level that is expected to be enabled on the card. HP-DLPI provides the promiscuous level on the instance by taking into account:  
• All the promiscuous requests from all streams attached to the interface.  
• The promiscuous level implied when the number of unique multicast enabled on the interface exceeds the NIC/driver specified limit.

DLPI_PROP_PROTOCOL_INFO  
Reserved for future use.
DLPI_PROP_VLAN_INFOP

dlpi_hdlp: Must be the handle provided by HP-DLPI.

opcode: DL_HP_OP_GET

valuep:

In: Pointer to dl_hp_getinfo_t.

Out: Fills up the dl_hp_getinfo_t structure and sets the dhg_datap element to point to a memory area containing an array of VLANs created on the interface. Each element of the array is of type dl_hp_vlan_instance_info_t.

This property is valid for tightly coupled drivers. This property can be used to get the list of VLANs enabled on an instance whose HP-DLPI handle is dlpi_hdlp.

HP-DLPI will allocate the dhg_datap element of the dl_hp_getinfo_t structure.

The dhg_datap element of the dl_hp_getinfo_t structure must be freed using the dhg_free_funcp routine provided in the same structure. No locks must be held across the call to dhg_free_funcp.

DLPI_PROP_OOP_INFOP

dlpi_hdlp: Must be the HP-DLPI provided handle.

opcode: DL_HP_OP_GET

valuep:

In: Pointer to dl_hp_oop_template_t.

Out: Filled structure of type dl_hp_oop_template_t. HP-DLPI will set the dhot_mblkp element of the structure with the OOP template.

This property is valid only for tightly coupled drivers.

This property must be used by drivers that support out-of-packet data on the inbound side. See dl_hp_drv_features_one_t(9F) for information on which drivers must use the DLPI_PROP_OOP_INFOP. HP-DLPI will calculate the maximum out-of-packet data size which the driver must to use to allocate additional size for each inbound packet and also the template of the OOP data.

The driver can provide the padding space required in the OOP template to avoid IP pull-ups (4 byte aligned IP header)

The MBLK provided here by HP-DLPI must not be freed by the driver at any time.

If a driver supports outbound OOP but does not support inbound OOP, for loopback packets the driver needs to move the mblk read pointer to point to the actual payload before looping back the packets.

See the HP-UX 11i v2 Driver Development Guide for information on how to implement OOP in a HP-DLPI based network interface driver.

DLPI_PROP_INSTANCE-INFOP

dlpi_hdlp: Must be NULL.

opcode: DL_HP_OP_GET
valuep:

In: Pointer to dl_hp_getinfo_t.

Out: Filled structure of type dl_hp_getinfo_t. HP-DLPI will set the dhg_datap element of dl_hp_getinfo_t to a memory area that contains multiple entries of type dl_hp_instance_info_t.

Search and get interface information about all interfaces that match the criteria specified in the dhg_modifier field of the dl_hp_getinfo_t structure passed via the valuep parameter.

On a successful return, the dhg_datap field of the dl_hp_getinfo_t structure passed in valuep will contain information about all the ports that match the criteria.

The information present in dhg_datap will be an array of type dl_hp_instance_info_t.

DLPI_PROP_MODULEP Reserved for internal use.
DLPI_PROP_MOD_ACQUIRE_PORT Reserved for internal use.
DLPI_PROP_MOD_RESERVED Reserved for internal use.
DLPI_PROP_MOD_RELEASE_PORT Reserved for internal use.
DLPI_PROP_MOD_ASSOCIATE_PORT Reserved for Internal use.
DLPI_PROP_MOD_DISASSOCIATE_PORT Reserved for internal use.
DLPI_PROP_MOD_UPDATE_INSTANCEP Reserved for internal use.
DLPI_PROP_MOD_CONTROL Reserved for internal use.
DLPI_PROP_MOD_PPA_RANGE Reserved for future use.
DLPI_PROP_RESERVED2 Reserved for future use.
DLPI_PROP_OP_PARAM dlp_hdlp: Must be the HP-DLPI provided per-interface handle.

opcode: DL_HP_OP_SET

valuep:

In: Pointer to dl_hp_op_param_t.

Description:

This property is valid for tightly coupled and loosely coupled drivers.

This property will set any one or more of the following values: Speed, Duplex, autoneg/auto sense/MTU.

The speed must be given in Megabits per second (Mb/s).

The MTU must be given in bytes. The minimum MTU must be 10; the minimum recommended MTU is 256.

Duplex value is of type dl_hp_duplex_mode_t.

Autoneg/Autosense value is of type dl_hp_autoneg_sense_t.

Driver must generate events if any of the operating parameters giving in this structure change. See the HP-UX 11i v2 Driver Development Guide for more information.
**DLPI_PROP_OOP_OUT_INFOP**

*dlpi_hdlp*: Must be an HP-DLPI provided per-interface handle.

*opcode*: DL_HP_OP_SET

*valuep*:

**In**: Pointer to `dl_hp_oop_out_template_t`.

This property is valid only for tightly coupled drivers. This property must be used by the drivers that expect upper layers to pass additional OOP information that HP-DLPI does not understand. The template that is passed as part of this property is included in the outbound template that HP-DLPI creates for HP-DLPI users.

**SEE ALSO**

`dl_hp_special_params_t(9S)`

**DLPI_PROP_RESERVED3**

Reserved for internal use.

**DLPI_PROP_TARP_OPTZER**

Reserved for internal use.

**DLPI_PROP_IFLASTCHANGE**

*dlpi_hdlp*: Must be the HP-DLPI provided handle.

*opcode*: DL_HP_OP_GET

*valuep*:

**In**: Pointer to `TimeTicks`.

**Out**: Filled with a time value in increments of 10 milliseconds.

This property is for tightly coupled drivers. This property must be used by drivers to set its internal `ifLastChange` element of the MIB II structure. This value must be directly assigned to `ifLastChange`. Drivers must set `ifLastChange` whenever there is a change in its `ifOper` state.

**DLPI_PROP_DRIVER_INFOP**

Reserved for internal use.

**DLPI_PROP_MODULE_INFOP**

Reserved for internal use.
**NAME**

DL_HP_RESET_STATS_REQ - Request to reset physical interface statistics.

**SYNOPSIS**

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

**DESCRIPTION**

This primitive is sent to drivers using the `driver_controlp` routine to reset the physical interface statistics. The associated structure for this request is the `dl_hp_reset_stats_req_t` structure.

The `DL_HP_RESET_STATS_REQ` primitive described in this manpage might be different from the definition in the *DLPI Programmer's Guide*. The information provided here is applicable only to tightly coupled drivers.

**STRUCTURE MEMBERS**

```c
typedef struct {
    uint32_t dl_primitive;
} dl_hp_reset_stats_req_t;
```

**CONSTRAINTS**

Drivers must call HP-DLPI event handler with `DL_HP_EVENT_CLEAR_STATS` event to reset per-instance HP-DLPI level statistics after resetting its internal statistics. See `dlpi_eventp(9F)` for more information.

**SEE ALSO**

`driver_controlp(9F), dlpi_eventp(9F)`
NAME
dl_hp_search_info_t - Search criteria to search the interface list.

SYNOPSIS
#include <sio/dlpi_drv.h>

DESCRIPTION
Specification for searching the interface list. Used when invoking the
DLPI_PROP_INSTANCE_INFO properties.

STRUCTURE MEMBERS
typedef struct dl_hp_search_info {
    dl_hp_search_type_t             dhsi_match;
    uint32_t                       dhsi_ppa;
    char                           *dhsi_drv_namep;
    dl_hp_drv_features_one_t       dhsi_features;
    uint32_t                        dhsi_features_two;
    uint32_t                        dhsi_features_three;
    dl_hp_mac_type_t               dhsi_mac_type;
    void                            *dhsi_rsvdp;
} dl_hp_search_info_t;

dhsi_match
    Specifies which of the following fields are to be matched during
    the search of the global interface list. Multiple fields could be
    specified and the search will match all the specified criteria. See
dl_hp_search_type_t(9F) for more information.

dhsi_ppa
    PPA to search for.

dhsi_drv_namep
    Get all instances of this driver that matches the driver name passed.
The name must be less than or equal to 64 characters and must be
NULL terminated. The name is case sensitive.

dhsi_features
    Match all the interfaces that have the same set or a super-set of
    these features. See dl_hp_drv_features_one_t(9F) for more
    information.

dhsi_features_two
    Reserved for future use. Must be set to 0.

dhsi_features_three
    Reserved for future use. Must be set to 0.

dhsi_mac_type
    Match all the interfaces that have the MAC type given. See
dl_hp_mac_type_t(9F) for additional information.

dhsi_rsvdp
    Reserved for future use. Must be set to 0.

SEE ALSO
dlpiprop(9F), dl_hp_features_one_t(9S), dl_hp_mac_type_t(9S)
NAME

dl_hp_search_type_t - Criteria for searching the interface list.

SYNOPSIS

#include <sio/dlpi_drv.h>

DESCRIPTION

Specifications for searching the interface list. This is a bit wise enumerated type.

STRUCTURE MEMBERS

typedef enum dl_hp_search_type {
    DL_HP_MATCH_PPA,
    DL_HP_MATCH_DRIVER_NAME,
    DL_HP_MATCH_FEATURES,
    DL_HP_MATCH_MAC_TYPE
} dl_hp_search_type_t

DL_HP_MATCH_PPA Match the PPA given in the dhsi_ppa element of
dl_hp_search_info_t structure.

DL_HP_MATCH_DRIVER_NAME Match the driver name given in the dhsi_drv_namep
element of dl_hp_search_info_t structure.

DL_HP_MATCH_FEATURES Match the features given in the dhsi_features element
of dl_hp_search_info_t structure.

DL_HP_MATCH_MAC_TYPE Match the MAC type given in the dhsi_mac_type element
of dl_hp_search_info_t structure.

SEE ALSO

dl_hp_search_info_t(9S)
NAME

DL_HP_SET_DRV_PARAM_IOCTL - HP-DLPI defined transparent IOCTL to set driver parameters.

SYNOPSIS

#include <sys/dlpi_ext.h>

DESCRIPTION

This IOCTL is sent to drivers using the driver_controlp routine. The associated structure for this request is dl_hp_set_drv_param_ioctl_t structure.

This IOCTL is defined by HP-DLPI for usage by drivers. This IOCTL is transparent to HP-DLPI (there are few exceptions, see the CONSTRAINTS section). Transparent IOCTLs are the ones that are not interpreted by HP-DLPI and are sent directly to the driver.

Use this IOCTL to set driver operating parameters like speed, duplex mode, MTU, and autonegotiation. Drivers can add new sub-requests to satisfy any driver specific commands. See the HP-UX 11i v2 Driver Development Guide for more information on defining sub-requests.

STRUCTURE MEMBERS

typedef struct dl_hp_set_drv_param_ioctl {
    dl_hp_drv_param_req_type_t dl_request;
    uint32_t dl_speed;
    dl_hp_duplex_mode_t dl_duplex;
    dl_hp_autoneg_sense_t dl_autoneg;
    uint32_t dl_mtu;
    uint64_t dl_value1;
    uint64_t dl_value2;
    uint64_t dl_value3;
    uint32_t dl_reserved1[2];
    uint32_t dl_reserved2[2];
} dl_hp_set_drv_param_ioctl_t;

- **dl_request**: The sub-request. The driver property(s) that need to be set. See dl_hp_drv_param_req_type_t(9F) for more information.
- **dl_speed**: Set the speed of underlying link in Megabits per second (Mb/s).
- **dl_duplex**: Set full or half duplex modes. The field is of type dl_hp_duplex_mode_t.
- **dl_autoneg**: Set the auto-negotiation/auto-sense. The field is of type dl_hp_autoneg_sense_t.
- **dl_mtu**: Set the MTU of the interface in octets.
- **dl_value1**: 64-bit value for driver extension.
- **dl_value2**: 64-bit value for driver extension.
- **dl_value3**: 64-bit value for driver extension.
- **dl_reserved1[2]**: 32-bit values for driver extension.
- **dl_reserved2[2]**: Reserved for internal use by HP-DLPI. Must be set to 0.

CONSTRAINTS

HP-DLPI does not interpret this request (it is transparent) except for the DL_HP_SERIALIZE bit. This request must be used between the driver’s user space application and the driver.
The driver user space component can request HP-DLPI to serialize this request when sending it to the driver, if another DL_HP_SET_DRV_PARAM_IOCTL on the same interface is in progress, by setting the DL_HP_SERIALIZE bit in the dl_request field. If serialization is requested, DLPI will wait for the in-progress DL_HP_SET_DRV_PARAM_IOCTL to complete before sending another.

The drivers can define their own dl_request by using any of the bits reserved for driver's extension or defining new bits that are not defined by HP-DLPI. The reserved bits are defined in dl_hp_drv_param_req_type_t(9F).

The driver must check the privileges of the user who issued the request (using the ioc_cr field of iocblk).

The driver must fill up the ioc_count value in the IOCTL MBLK if the request is successfully processed.

If the IOCTL fails, HP-DLPI will free the b_cont of the MBLK that contains the data associated with the IOCTL.

SEE ALSO
dli_controlp(9F), dl_hp_drv_param_req_type_t(9S), dl_hp_duplex_mode_t(9S).
dl_hp_autoneg_sense_t(9S)
NAME
DL_HP_SET_IFADMIN_REQ - Request to set the physical interface administrative status.

SYNOPSIS
#include <sys/dlpi_ext.h>

DESCRIPTION
This primitive is sent to drivers using the driver_controlp routine to set the administrative state of the interface. The associated structure for this request is dl_hp_set_ifadmin_req_t structure.

This primitive is applicable only between HP-DLPI and the driver.

STRUCTURE MEMBERS
typedef struct dl_hp_set_ifadmin_req {
    uint32_t          dl_primitive;
    dl_hp_ifadmin_state_t dl_ifAdminStatus;
    uint32_t          dl_reserved;
} dl_hp_set_ifadmin_req_t;

dl_primitive        Set to DL_HP_SET_IFADMIN_REQ.
dl_ifAdminStatus    The ifAdmin value to be set on the interface. The element is of type dl_hp_ifadmin_state_t.
dl_reserved          Reserved for future use. Set to 0.

CONSTRAINTS
Requests driver to set the administrative status of the interface to the value specified in dl_ifAdminStatus.
Tightly coupled drivers must generate link DOWN and UP events when the ifAdmin status changes. See the HP-UX 11i v2 Driver Development Guide for more information.

SEE ALSO
driver_controlp(9F), dl_hp_ifadmin_state_t(9S)
**NAME**

`dl_hp_src_llc_info_t` - Exchange source information during LLC build header routine.

**SYNOPSIS**

```c
#include <sio/dlpi_drv.h>
```

**DESCRIPTION**

This structure is used by HP-DLPI to pass source information to the driver build header routine.

**STRUCTURE MEMBERS**

```c
typedef struct dl_hp_src_llc_info {
    uint8_t                   dsli_ssap;
    uint16_t                  dsli_sxsap;
    uint32_t                  dsli_dlsap_addr_length;
    uint8_t                   dsli_dlsap_addr[DL_HP_MAX_DLSAP_LEN];
    dl_hp_tagging_type_t      dsli_tagging_type;
    uint32_t                  dsli_pkt_length;
    uint32_t                  dsli_ip_stream;
    uint32_t                  dsli_arp_stream;
    uint32_t                  dsli_no_src_routing;
    uint8_t                   dsli_ctrl;
    uint64_t                  dsli_reserved1[2];
    uint64_t                  dsli_reserved2[2];
} dl_hp_src_llc_info_t;
```

- `dsli_ssap` Identifies the source SAP. Depending on the type of packet (XID/TEST Response or request) the LSB of the SAP value will be set. The build header routine must use the SAP value from this element instead of extracting the value from `dsli_dlsap_addr`.
- `dsli_sxsap` Identifies the extended source SAP. You can extract this value from `dsli_dlsap_addr` as well.
- `dsli_dlsap_addr_length` Identifies the DLSAP address length in bytes.
- `dsli_dlsap_addr` Identifies the source DLSAP address. See the HP-UX 11i v2 Driver Development Guide for more information.
- `dsli_tagging_type` Identifies whether the VLAN packet is for a driver that supports software-based VLAN tagging or hardware-based VLAN tagging. For an enumeration of supported tagging types, see `dl_hp_tagging_type_t`(9S).
- `dsli_pkt_length` Identifies the packet length in bytes. This is the data length of the packet (excluding the LLC header length). Drivers must add to this length field before filling the length for IEEE802.3 and SNAP802.3 packets.
- `dsli_ip_stream` Set to 1 if the packet is an IPv4 or IPv6 packet. This helps the driver’s build header routine in understanding the format of RIF information.
- `dsli_arp_stream` Set to 1 if the packet is an ARP packet. This helps the driver’s build header routine in understanding the format of RIF information.
dsli_no_src_routing

Set to 1 if Source routing is disabled (only for Token Ring packets).

Identifies the control type.

Reserved for future use. It must be set to 0.

Reserved for future use. It must be set to 0.

SEE ALSO

dl_hp_tagging_type_t(9S)
DL_PHYS_ADDR_REQ(9G)

NAME

DL_PHYS_ADDR_REQ - Request to get physical address (MAC address).

SYNOPSIS

#include <sys/dlpi.h>

DESCRIPTION

This primitive is sent to drivers using the driver_controlp routine to retrieve physical address of the card. The request could be to retrieve factory default address or the current physical address. The request information is present in the dl_phys_addr_req_t structure.

The DL_PHYS_ADDR_REQ primitive described in this manpage may be different from the definition in the DLPI Programmer's Guide. The information provided here is applicable only to tightly coupled drivers.

STRUCTURE MEMBERS

typedef struct {
    uint32_t    dl_primitive;
    uint32_t    dl_addr_type;
    uint32_t    dl_addr_offset;
} dl_phys_addr_req_t;

dl_primitive Set to DL_PHYS_ADDR_REQ

dl_addr_type Type of physical address requested:

    DL_FACT_PHYS_ADDR — To retrieve factory default physical address.
    DL_CURR_PHYS_ADDR — To retrieve current physical address.

dl_addr_offset Offset from the beginning of the message block (MBLK->b_rptr) where the physical address begins.

CONSTRAINTS

The second MBLK (linked by b_cont) contains space to hold the physical address. The second MBLK is empty when HP-DLPI passes it to the driver (b_rptr == b_wptr). After writing the information to the second MBLK, drivers must move the b_wptr to the size that was written to the second MBLK (in this case by the size of the physical address).

HP-DLPI handles MAC addresses in canonical format for Ethernet and FDDI drivers and wire format for Token Ring drivers.

SEE ALSO

driver_controlp(9F)
**NAME**

DL_PROMISCOFF_REQ - Request to disable a promiscuous level on the physical interface.

**SYNOPSIS**

#include <sys/dlpi.h>

**DESCRIPTION**

This primitive is sent to drivers using the driver_controlp routine to disable promiscuous mode on the physical interface. The associated structure for this request is the dl_promiscoff_req_t structure.

The DL_PROMISCOFF_REQ primitive described in this manpage may be different from the definition in the DLPI Programmer’s Guide. The information provided here is applicable only to tightly coupled drivers.

**STRUCTURE MEMBERS**

typedef struct {
    uint32_t dlPrimitive;
    uint32_t dlLevel;
} dl_promiscoff_req_t;

dlPrimitive Set to DL_PROMISCOFF_REQ

**CONSTRAINTS**

The dlLevel parameter must be ignored by the driver and must disable all the promiscuous levels enabled on the card.

HP-DLPI will always succeed the DL_PROMISCOFF_REQ request to the user/application, even if driver failed in its processing. If the driver fails the request, it will go out-of-sync (the promiscuous level in the driver will be different from what is maintained in HP-DLPI) with HP-DLPI. To recover from this situation, the driver can attempt to reprogram the promiscuous mode (by getting them from HP-DLPI using a property) at a convenient time. Failure to do so, will result in performance degradation.

If this primitive is received when the card is in physical or multicast promiscuous mode, the driver must disable all promiscuous levels. In addition, the driver must get the multicast list from HP-DLPI using the driver_propp call and reprogram the card.

**SEE ALSO**

driver_controlp(9F), dlpi_propp(9F)
**NAME**

DL_PROMISCON_REQ - Request to enable a promiscuous level on the physical interface.

**SYNOPSIS**

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

**DESCRIPTION**

This primitive is sent to drivers using the `driver_controlp` routine to enable a level of promiscuous mode on the interface. The associated structure for this request is the `dl_promiscon_req_t` structure.

The `DL_PROMISCON_REQ` primitive described in this manpage may be different from the definition in the *DLPI Programmer's Guide*. The information provided here is applicable only to tightly coupled drivers.

**STRUCTURE MEMBERS**

```c
typedef struct {
    uint32_t dl_primitive;
    uint32_t dl_level;
} dl_promiscon_req_t;
```

- `dl_primitive` Set to `DL_PROMISCON_REQ`
- `dl_level` Indicates promiscuous level. The valid values for the promiscuous level are:
  - `DL_PROMISC_PHYS` — Physical promiscuous mode.
  - `DL_PROMISC_SAP` — SAP promiscuous mode.
  - `DL_PROMISC_MULTI` — Multicast promiscuous mode.

**CONSTRAINTS**

If the HP-DLPI requested promiscuous level in `dl_level` is different from the level that is enabled in the card, the driver can disable the current level and enable the level requested by HP-DLPI.

If the promiscuous level changes to `DL_PROMISC_SAP` from a higher level (`DL_PROMISC_MULTI` or `DL_PROMISC_PHYS`), the driver must get the list of multicast addresses from HP-DLPI using `driver_propp` and reprogram the card.

HP-DLPI will send the `DL_PROMISCON_REQ` primitive only if a user has requested higher promiscuous level than the card is already in.

**SEE ALSO**

`driver_controlp(9F), dlpi_propp(9F)`
NAME

DL_SET_PHYS_ADDR_REQ - Request to set physical address (MAC address).

SYNOPSIS

#include <sys/dlpi.h>

DESCRIPTION

This primitive is sent to drivers using the driver_controlp routine to set the physical address of the card. The request is to change the physical interface address. The request information is present in the dl_set_phys_addr_req_t structure.

The DL_SET_PHYS_ADDR_REQ primitive described here might be different from the definition in the DLPI Programmer's Guide. The information provided here is applicable only to tightly coupled drivers.

STRUCTURE MEMBERS

typedef struct {
    uint32_t    dl_primitive;
    uint32_t    dl_addr_length;
    uint32_t    dl_addr_offset;
} dl_phys_addr_req_t;

dl_primitive  Set to DL_SET_PHYS_ADDR_REQ

dl_addr_length Length of the physical address.

dl_addr_offset Offset from the beginning of the message block (b_rptr of the message block) where the multicast address begins.

CONSTRAINTS

MAC address passed to the driver will in canonical format for Ethernet drivers.

Drivers need to generate link events when there is a change in physical address. See the HP-UX 11i v2 Driver Development Guide for more information on events.

SEE ALSO

driver_controlp(9F)
dlpi_eventp(9F)

NAME

dlpi_eventp - HP-DLPI event handler.

SYNOPSIS

#include <sio/dlpi_drv.h>

void (*dlpi_eventp)(
    void *dlpi_hdlp,
    dl_hp_event_type_t event,
    void *event_infop,
    void *rsvdp
);

PARAMETERS

dlpi_hdlp Pointer to HP-DLPI private handle (opaque to drivers) that was obtained during initialization as part of the DL_HP_OP_CREATE operation.

event HP-DLPI event to process. The element is of type dl_hp_event_type_t.

event_infop Associated information for the event.

rsvdp Reserved for future use. Must be set to NULL.

DESCRIPTION

This is the HP-DLPI event handler. This interface is applicable only for tightly coupled drivers. The event handler will be used to notify HP-DLPI of link events (such as state changes, clear HP-DLPI's internal statistics, and so on). The event handler routine will take additional hints or information, if required. For events like link DOWN and UP, drivers must pass the hint which specifies why the event happened. For example, when driver sends a link DOWN event during the MAC address change, it will pass the information in the event_infop parameter. For link UP events, a driver can use DL_HP_EVENT_DEFAULT if the actual cause of link UP event is not known (see dl_hp_event_link_cause_t(9S)).

RETURN VALUES

None

CONSTRAINTS

This routine must be requested from HP-DLPI using the driver_propp routine. The DLPI_PROP_EVENTP property will return the function pointer to this routine (see dl_hp_prop_t(9S)).

No locks must be held while calling this routine.

HP-DLPI does not guarantee the synchronous execution of events in the same thread, but guarantees the order of processing of the events (the events will be processed in the order they are received).

WARNINGS

None

EXAMPLES

SEE ALSO

dlpi_propp(9F), dl_hp_prop_t(9S), dl_hp_event_link_cause_t(9S), dl_hp_event_type_t(9S)
NAME
dlpi_inboundp - HP-DLPI inbound packet processing routine.

SYNOPSIS
#include <sio/dlpi_drv.h>
void (*dlpi_inboundp)(
    void *dlpi_hdlp,
    mblk_t *mblkp,
    void *rsvd1p,
    void *rsvd2p
) ;

PARAMETERS
dlpi_hdlp Pointer to HP-DLPI private handle (opaque to drivers) that was obtained during the DL_HP_OP_CREATE operation.
mblkp Inbound datagram.
rsvd1p Reserved for future use. Must be set to NULL.
rsvd1p Reserved for future use. Must be set to NULL

DESCRIPTION
Inbound path for all packet formats. HP-DLPI provides inbound routines for Ethernet technologies.

RETURN VALUES
None

CONSTRAINTS
The inbound routine must be requested from HP-DLPI using the dlpi_propp interface. The property to use is DLPI_PROP_INBOUNDP.

No locks must be held when calling the inbound routine.

The out-of-packet data template created by HP-DLPI during driver's call to dlpi_propp with DLPI_PROP_OOP_INFOP property must be used to pass any OOP information. See the HP-UX 11i v2 Driver Development Guide for HP-DLPI to Driver options negotiations and out-of-packet data passing mechanism.

The b_rptr of the MBLK must either point to the OOP if it is present or to the MAC header (that is, the destination address for Ethernet).

If OOP is present, the b_flag of the MBLK containing the OOP must have DL_HP_OOP_PRESENT flag set, see the HP-UX 11i v2 Driver Development Guide for more information.

The beginning of the MBLK data (b_rptr) must be 4 byte aligned.

When the driver is looping back an outbound packet, the driver must also pass the OOP information that was sent down by HP-DLPI.

Drivers that handle packet trains, must copy the cko_offset element of cko_info_t structure and the VLAN Tag information from the first fragment of the packet train to the subsequent fragment that is being looped back. The fragments must be separated and sent as individual packets to HP-DLPI. See the HP-UX 11i v2 Driver Development Guide for more information.

For loopback 802.3 packet trains, the drivers must also calculate the length of each one of the packet fragments and place it in the length part of LLC header before sending to HP-DLPI. The
reason is when a packet train is sent to the driver, HP-DLPI will update the length part of the 802.2 header only for the first fragment of the packet train.

HP-DLPI handles automatic response for SAP 0 TEST packets.

SEE ALSO

dlpi_propp(9F)
NAME

dlpi_propp - HP-DLPI property interface

SYNOPSIS

#include <sio/dlpi_drv.h>

int32_t (*dlpi_propp)(
    void *dlpi_hdlp,
    dl_hp_op_t opcode,
    dl_hp_prop_t prop_name,
    void *valuep,
    void *rsvd1p,
    void *rsvd2p);

PARAMETERS

dlpi_hdlp
Pointer to HP-DLPI private handle (opaque to drivers) that was obtained during
the DL_HP_OP_CREATE operation (dlpi_hdlp must be set to NULL for the
DL_HP_OP_CREATE operation itself). It is possible to pass NULL for dlpi_hdlp
when calling dlpi_propp for certain properties. See dl_hp_prop_t(9F) for more
information.

opcode
Enumerated type (dl_hp_op_t) specifying the operation.
DL_HP_OP_CREATE Create a new instance of HP-DLPI per-instance
structures.
DL_HP_OP_DELETE Delete data structures pertaining to this instance.
DL_HP_OP_GET Retrieve a property.
DL_HP_OP_SET Set a property in HP-DLPI.
Support for an opcode depends on the property to which it is applied.

prop_name
Property to Create/Delete/Get/Set. Parameter is of type dl_hp_prop_t.
valuep
For DL_HP_OP_GET, this is an “in” and “out” parameter.
For DL_HP_OP_SET, this is an “in” parameter. This parameter specifies the value
associated with the property.

rsvd1p
Reserved for future use. Must be set to NULL.

rsvd2p
Reserved for future use. Must be set to NULL.

DESCRIPTION

This interface will be provided by HP-DLPI and must be used by all drivers to exchange
interface-specific information between HP-DLPI and the driver. In general, a pull approach is
favored; the module that needs the data pulls it from the module supplying the data by calling
the supplier's property function. Exceptions, where they occur, are noted.

The property routine is a well-known global function pointer. The syntax of the function will
not change once a driver or module is written to use it.

RETURN VALUES

0 for success.
EBUSY HP-DLPI cannot handle this property at this time.
EINVAL Invalid property call. For example, loosely coupled drivers doing a get on the HP-DLPI event handler or the HP-DLPI inbound routine.

ENOBUS Allocation of message blocks failed.

ENOENT No entries found.

ENOMEM Memory allocation failure.

ENOTSUP Unsupported property or operation.

CONSTRAINTS

No locks must be held while calling into this routine.
Call to dlpi_propp will not block or sleep.

Caller must pass the buffer for the structure associated with a DL_HP_OP_GET property. Any additional allocations required to satisfy the request is allocated by HP-DLPI and must be freed by the driver using the free routine returned along with the information returned. There are few exceptions, see dl_hp_prop_t(9F) for the individual property definitions.

HP-DLPI private handle must be passed for all the dlpi_propp calls (see dl_hp_prop_t(9F) for exceptions). The DL_HP_OP_CREATE operation will return the HP-DLPI private handle to the caller.

The DL_HP_OP_DELETE operation for tightly-coupled physical drivers is supported only when the initialization of the driver instance fails. After initialization, the driver instance cannot be deleted. Deletion of loosely-coupled driver instances is allowed anytime.

HP-DLPI maintains a copy of some of the driver properties like Speed, MTU, ifAdmin status for the purpose of giving it to other modules that depend on them. Such properties must be set by the driver using the list of DL_HP_OP_SET properties listed in dl_hp_prop_t(9F). The driver is responsible for maintaining the copy of driver properties in HP-DLPI.

HP-DLPI recommends that property calls be avoided in performance paths as they are not performance-friendly.

If a property call fails, all the actions that are supposed to happen in that property has failed. In other words, no partial processing of a property call is supported.

The values associated with the properties can be categorized into “In” and “Out” values depending on whether the value is passed to HP-DLPI by the caller or HP-DLPI returns the value to the callee. “In” means that content of the valuep parameter is passed by the driver to HP-DLPI. “Out” means the value is passed back from HP-DLPI to the driver. “In/Out” means, the information can be passed in by the driver while calling HP-DLPI and, HP-DLPI can pass back information when the call returns.

SEE ALSO

dl_hp_prop_t(9S)
**NAME**

dlpi_wakeupp - HP-DLPI interface to be used by drivers to notify the asynchronous completion of control requests.

**SYNOPSIS**

```c
#include <sio/dlpi_drv.h>
void (*dlpi_wakeupp)(
   void *dlpi_hdlp,
   mblk_t *mblkp,
   uint32_t status,
   void *rsvdp,
);
```

**PARAMETERS**

- **dlpi_hdlp**: Pointer to HP-DLPI private handle (opaque to drivers) that was obtained during initialization as part of the DL_HP_OP_CREATE operation.
- **mblkp**: The address of the MBLK containing the request that was passed to the driver (see `driver_controlp(9F)`).
- **status**: Status of the control request for which the driver is doing the wakeup.
- **rsvdp**: Reserved for future use. Must be set to NULL.

**DESCRIPTION**

HP-DLPI provides a wakeup handler for tightly coupled drivers to do an asynchronous notification of the completion of a control request HP-DLPI sent to the driver. All the requests sent to HP-DLPI using `driver_controlp` will require a wakeup. Drivers will be able to return errors for all the control requests in the wakeup handler routine.

**RETURN VALUES**

None

**CONSTRAINTS**

This routine must be requested from HP-DLPI during the initial property exchange. The property to use is DLPI_PROP_WAKEUPP (see `dl_hp_prop_t(9F)`).

No locks must be held when calling this routine.

Driver must guarantee that all the control requests are woken up. Failure to do so may cause subsequent control requests to be blocked by DLPI.

The drivers must not issue redundant wakeup for a control request. Even though HP-DLPI has mechanisms to detect this kind of situation, it may potentially cause erroneous behavior.

The `dlpi_wakeupp` call may happen before the `driver_controlp` call completes, but driver must not access the control request MBLK after the wakeup call has been made.

**SEE ALSO**

`driver_controlp(9F), dlpi_wakeupp(9F), driver_controlp(9F)`
NAME

driver_build_headerp - Driver packet build header routine.

SYNOPSIS

#include <sio/dlpi_drv.h>

uint32_t (*driver_build_headerp)(
    void *driver_hdlp,
    dl_hp_src_llc_info_t *source_infop,
    dl_hp_dest_llc_info_t *dest_infop,
    dl_hp_llc_info_t *llc_hdrp,
    void *rsvd1p,
    void *rsvd2p
);

PARAMETERS

driver_hdlp    Pointer to opaque instance specific control handle given to HP-DLPI by the driver during initialization. HP-DLPI passes this to driver when calling driver. This is typically driver's interface structure.

source_infop  This contains the source information for the packet including source SAP, source address, source extended SAP, VLAN information, and RIF information. This is used to construct the LLC header. This element will be of type dl_hp_src_llc_info_t.

dest_infop    The destination information, such as dsap, dxsap, and destination addr. Parameter is of type dl_hp_dest_llc_info_t.

llc_hdrp      Pointer to the structure containing the LLC header information. HP-DLPI will allocate this parameter. The dli_mblkp element of dl_hp_llc_info_t structure is allocated and built by the build header routine. The structure will be of type dl_hp_llc_info_t.

rsvd1p        Reserved for future use. Set to NULL.

rsvd2p        Reserved for future use. Set to NULL.

DESCRIPTION

The purpose of the build header routine is to create the MAC and LLC header for outbound packets. This routine will also be called during fastpath header template negotiations (see the HP-UX 11i v2 Driver Development Guide) with transport to create the MAC and LLC header template. HP-DLPI will provide a default build header routine for Ethernet drivers if they do not provide their own build header routine. This driver build header entry point is provided by driver during the property exchange time (see dlpi_propp(9F)) as part of the DL_HP_OP_CREATE operation. If the routine provided is NULL, HP-DLPI uses HP-DLPI's build header routine.

RETURN VALUES

0       Success.
EINVAL  Header cannot be created because of invalid values in elements of source_infop and dest_infop have incorrectly formatted. Set llc_hdrp to NULL.
ENOBUFS  Failure to allocate message buffers.

CONSTRAINTS

No locks will be held by HP-DLPI when calling the build header routine.
This interface is optional. HP-DLPI will use its internal build header routine to construct Ethernet packet headers.

This interface may be provided by the driver to HP-DLPI during property exchange as `dhc_build_hdrp` element of `dl_hp_create_info_t` structure (see `dlpi_propp(9F)`).

If the driver provides a build header routine it will be requested to build the link level header. HP-DLPI will handle MAC and multicast addresses in canonical format only for Ethernet drivers. If the driver build header routine finds an error in the information passed or the build header processing is not successful, it must return NULL in the `dli_mblkp` element of `dl_hp_llc_info_t` pointer and must return an error value for this routine.

Drivers are not allowed to block or sleep during this call.

**SEE ALSO**

`dlpi_propp(9F), dl_hp_src_llc_info_t(9S), dl_hp_dest_llc_info_t(9S), dl_hp_llc_info_t(9S), dl_hp_create_info_t(9S)`
NAME

driver_controlp - Driver control request handler.

SYNOPSIS

#include <sio/dlpi_drv.h>

void (*driver_controlp)(
    void *driver_hdlp,
    int32_t cmd,
    mblk_t *mblkp,
    void *rsvdp
);

PARAMETERS

driver_hdlp  Pointer to opaque instance specific control handle given to HP-DLPI by the
driver during initialization as dhc_control_hdlp in
dl_hp_create_info_t. HP-DLPI passes this handle to the driver when
calling driver's control routine. This is typically a driver's per-instance
structure.

cmd          A 32-bit integer command/request value.

mblkp        Pointer to MBLK containing the data associated with the request. See the
DESCRIPTION section for the list of requests that will be passed to the driver.
The request MBLK passed must not be modified by the driver.

rsvdp        Reserved for future use. Set to NULL.

DESCRIPTION

Tightly coupled drivers provide a control request handler that the HP-DLPI module calls to pass
any requests to the drivers. Drivers, for all the control requests irrespective of whether they
support or not, must do a wakeup using the dlpi_wakeupp interface. See the HP-UX 11i v2
Driver Development Guide for more information on HP-DLPI's simulated sleep/wakeup mechanism
and the control requests and kinds of information that will be passed along with the control
requests.

For certain requests, HP-DLPI will allocate the required size to hold the information that is
requested from the driver and link it to the b_cont of the request MBLK passed. The driver will
copy the required information depending on the request and pass it back during the wakeup of
HP-DLPI by calling the wakeup function. HP-DLPI will then do rest of the processing of
constructing the acknowledgement MBLK before responding to the user. The IOCTLs are
transparent to HP-DLPI. HP-DLPI passes the IOCTLs down to the driver and expects the driver
to do all the processing, including allocation of additional space.

HP-DLPI requests contain the request information in the first message block and optionally a
second MBLK linked by b_cont of the request MBLK. The second MBLK is used to pass
information from the driver to HP-DLPI for requests such as DL_GET_STATISTICS_REQ.

IOCTLs requests are generally transparent to HP-DLPI, except for
DL_HP_SET_DRV_PARAM_IOCTL. See DL_HP_SET_DRV_PARAM_IOCTL(9F) for more
information). IOCTLs when passed to the driver will have first MBLK containing the iocblk
structure, which provides information to the driver such as the IOCTL command and the user
credentials. Following the first MBLK, linked by b_cont, is a second MBLK that contains any
IOCTL specific information.

The list of requests that will be passed to the driver are:

dl_enabmuliti_req  Request to enable specific multicast addresses.
DL_DISABMULTI_REQ  
Request to disable specific multicast addresses.

DL_PHYS_ADDR_REQ  
Request to get the physical address.

DL_SET_PHYS_ADDR_REQ  
Set the physical address to the value passed in this request.

DL_HP_HW_RESET_REQ  
Request to reset the physical interface.

DL_PROMISCOFF_REQ  
Request to disable promiscuous mode.

DL_PROMISCON_REQ  
Request to enable promiscuous mode.

DL_HP_RESET_STATS_REQ  
Request to reset the MIB statistics of an interface.

DL_GET_STATISTICS_REQ  
Request to get the standard and extended MIB statistics.

DL_HP_GET_MIBSTATS_REQ  
Request to get the standard and extended MIB statistics applicable to all interfaces.

DL_HP_GET_64BIT_STATS_REQ  
Request to get the 64-bit standard and extended MIB statistics applicable to those interfaces that support 64-bit statistics.

DL_HP_SET_IFADMIN_REQ  
Set the ifAdmin value.

DL_HP_SET_DRV_PARAM_IOCTL  
HP-DLPI defined transparent IOCTL to set driver parameters.

DL_HP_GET_DRV_PARAM_IOCTL  
HP-DLPI defined transparent IOCTL to get driver parameters.

See the respective manpages for additional information.

For an illustration of the flow of processing for these requests, see the HP-UX 11i v2 Driver Development Guide.

RETURN VALUES

None

CONSTRAINTS

All control requests received by this routine must be woken up. A missed wakeup call by the driver may result in future control requests being blocked. See dlpi_wakeupp(9F) for the wakeup processing routine.

Multiple control requests can be sent to the driver at the same time. Driver must take care of queueing and handling them. Drivers may choose to return error if they receive a control request while handling another request, but care must be taken to retain the request MBLK to wakeup HP-DLPI with error.

If the control request is to change the property of the interface (for example, MTU, speed, and MAC address), the drivers must inform HP-DLPI of the change using the dlpi_propp interface and may have to generate link DOWN and UP events. To find out how and when to generate link DOWN and UP events, see the HP-UX 11i v2 Driver Development Guide.

No locks will be held by HP-DLPI while calling this routine.

This interface must be provided by the driver to HP-DLPI during property exchange as dhc_controlp element of dl_hp_create_info_t structure.

Driver specific application and drivers can define IOCTLs that will be transparent to HP-DLPI. Transparent IOCTLs will be passed directly to the driver without any processing in HP-DLPI. For constraints on defining transparent IOCTLs, see the HP-UX 11i v2 Driver Development Guide.

For the definitions of IOCTLs, see the individual manpages.

HP-DLPI will allocate space for driver to fill the information for requests that require driver to pass information back to DLPI (for example, DL_GET_STATISTICS_REQ), see the individual request for this information.
See `dlpi_prop(9F)` for an explanation of driver control interface handle that will be passed as first parameter with every invocation of `dlpi_controlp` routine.

Drivers must not modify the control request MBLK (the first MBLK) in any way.

Drivers must not modify the `b_rptr` of the second MBLK (if there is a second MBLK).

Drivers are not allowed to block or sleep during this call.

For IOCTLs driver must do the necessary credential checks.

**WARNINGS**

None

**EXAMPLES**

**SEE ALSO**

`dlpi_prop(9F), dlpi_wakeupp(9F), dl_hp_create_info_t(9S), driver_controlp(9F), DL_ENABMULTI_REQ(9G), DL_DISABMULTI_REQ(9G), DL_SET_PHYS_ADDR_REQ(9G), DL_PHYS_ADDR_REQ(9G), DL_HP_HW_RESET_REQ(9G), DL_PROMISCOFF_REQ(9G), DL_PROMISCON_REQ(9G), DL_HP_RESET_STATS_REQ(9G), DL_GET_STATISTICS_REQ(9G), DL_HP_GET_MIBSTATS_REQ(9G), DL_HP_GET_64BIT_STATS_REQ(9G), DL_HP_SET_IFADMIN_REQ(9G), DL_HP_CREATE_VLAN_REQ(9G), DL_HP_DELETE_VLAN_REQ(9G), DL_HP_MODIFY_VLAN_REQ(9G), DL_HP_SET_DRV_PARAM_IOCTL(9G), DL_HP_GET_DRV_PARAM_IOCTL(9G)`
**NAME**

driver_event_handlerp - Driver event handler routine.

**SYNOPSIS**

```c
#include <sio/dlpi_drv.h>

int32_t (*driver_event_handlerp)(
    void *driver_hdlp,
    dl_hp_drv_event_type_t event_type,
    void *event_infop,
    void *rsvdp
);
```

**PARAMETERS**

- **driver_hdlp**  
  Pointer to opaque instance specific control handle given to HP-DLPI by the driver during initialization. HP-DLPI passes this to driver when calling driver. This is typically driver's interface structure.

- **event_type**  
  Event Type. See `dl_hp_drv_event_type_t(9F)`.

- **event_infop**  
  Any supplement information associated with this event.

- **rsvdp**  
  Reserved for future use. Set to NULL.

**DESCRIPTION**

This interface serves as an entry point into tightly coupled physical drivers to initiate some process in the driver. This event handler is by HP-DLPI.

**RETURN VALUES**

- **0**  
  Success.

- **ENXIO**  
  No such device.

- **ENOMEM**  
  Not enough kernel memory.

- **EINVAL**  
  One of the parameter has invalid value.

- **ENOBUFFS**  
  Not enough buffer space (message block).

- **ENOLINK**  
  Interface is not connected.

- **ENOTSUP**  
  Unsupported event.

**CONSTRAINTS**

This interface must be provided by the driver to HP-DLPI during property exchange as `dhc_eventp` element of `dl_hp_create_info_t` structure.

No locks will be held by HP-DLPI while calling this routine.

See `dlpi_propp(9F)` for an explanation of the driver event interface handle that will be passed as first parameter with every invocation of `driver_event_handlerp` routine.

Drivers are not allowed to block (sleep) during this call.

Any information passed via `event_infop` parameter will be invalid after the driver returns from the call. Drivers must copy the contents of `event_infop` if they need to reference them later.
WARNINGS
EXAMPLES
SEE ALSO
dlp1_propp(9F), dl_hp_drv_event_type_t(9S), driver_event_handlerp(9F), dl_hp_create_info_t(9S)
NAME

driver_outputp - Driver outbound packet processing routine.

SYNOPSIS

#include <sio/dlpi_DRV.h>

uint32_t (*driver_outputp)(
    void *driver_hdlp,
    mblk_t *mblkp,
    dl_hp_pkt_t pkt_type,
    void *rsvdp
); 

PARAMETERS

driver_hdlp Pointer to opaque instance specific control handle given to HP-DLPI by the
driver during initialization. HP-DLPI passes this to driver when calling driver.
This is typically driver's interface structure

mblkp Pointer to the data.

pkt_type Packet type. See dl_hp_pkt_type_t(9S).

rsvdp Reserved for future use. Set to NULL.

DESCRIPTION

This is the entry point for the drivers' data path. The driver output routine must be registered
with HP-DLPI during the driver property exchange (see dlpi_propp(9F)).
This routine applies only to tightly coupled drivers.

RETURN VALUES

0 Successfully enqueued the packet to the hardware transmit queue.
ENXIO No such device.
ENOMEM Not enough kernel memory.
EINVAL One of the parameter has invalid value.
ENOBUFFS Not enough buffer space (message block).
ENOLINK Interface is not connected.

CONSTRAINTS

Drivers must check for out-of-packet data by checking the b_flag bit for DL_HP_OOP_PRESENT
and process them. If the drivers want to ignore the OOP data, they can jump to the MAC/LLC
header based on the 4-byte offset (aligned at a word boundary) field that will be present in the
beginning of the packet. The 4-byte offset to the MAC/LLC header will be present only if
DL_OOP_PRESENT flag is set in the b_flag field. See the HP-UX 11i v2 Driver Development Guide
for more information on OOP data.
No locks will be held by HP-DLPI while calling this routine.
This interface must be provided by the driver to HP-DLPI during property exchange as
dhc_outputp element of dl_hp_create_info_t structure.
If drivers receive out-of-packet data, they must not assume the order of the data. They must
follow the guidelines described in the HP-UX 11i v2 Driver Development Guide.
Multicast, broadcast packets must be looped back if the MSGNOLOOP flag is not set in b_flag of
the MBLK. The packets must be looped back by doing the copymsg.
If promiscuous mode is enabled, the packets must be looped back irrespective of the MSGNOLOOP flag.

If driver loops back packets, it must update its inbound statistic counters.

HP-DLPI will not do any packet size checks against the driver MTU.

Packets received with pkt_type of DL_HP_RAW_PKT must not be modified by the driver.

See dlpi_propp(9F) for an explanation of the driver output interface handle that will be passed as first parameter with every invocation of the driver_outputp routine.

HP-DLPI will handle MAC and multicast addresses in canonical format for Ethernet and FDDI drivers and in wire format for Token Ring drivers.

Driver must not change the structure and order of the OOP data in the packets when the packets are looped-back.

Drivers that handle packet trains, must copy the cko_offset element of cko_info_t structure and the VLAN Tag information from the first fragment of the packet train to the subsequent fragment that is being looped back. See the HP-UX 11i v2 Driver Development Guide for more information.

The beginning of a new fragment in a packet train is specified by setting of the DL_HP_OOP_PRESENT flag in the b_flag of the MBLK.

Drivers must not change the Source Address of the packet.

Drivers are not allowed to block or sleep during this call.

**WARNINGS**

None

**EXAMPLES**

**SEE ALSO**

dlpi_propp(9F), dl_hp_pkt_type_t(9S), dl_hp_create_info_t(9S), copymsg(9F), driver_outputp(9F)
NAME
filter_packet - Checks if the packet matches filter criteria set up.

SYNOPSIS
#include <fmt.h>
#include <ntl.h>
#include <sys/subsys_id.h>
int filter_packet(void);

PARAMETERS
None

DESCRIPTION
For the current packet, checks all the filters set to see if the packet needs to be formatted. The
subformatter must call this routine after calling one of set_up_xxx routines, which sets up the
globals.

RETURN VALUES
0 Packet must not be formatted.
<> 0 Packet needs to be formatted.

CONSTRAINTS
None

SEE ALSO
set_up_8022(9F), set_up_ether(9F), set_up_link(9F), set_up_ip(9F), format_link_nice(9F),
format_link_raw(9F), format_link_terse(9F)
format_link_nice(9F)

NAME

format_link_nice - Formats the link level packet using nice formatting.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/subsys_id.h>

int format_link_nice(
    tl_msg_hdr_type *hdr,
    u_char *buffer,
    int len,
    char *linktype,
    char *line1,
    char *addlinfo,
    char *upperinfo
);

PARAMETERS

hdr Pointer to the standard nettl message header.
buffer Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer, but the uppermost layers may display data at an appropriate offset into the buffer.
len Length of the buffer (including 802.2, excluding any lower layer data).
linktype String describing the type of link this information is carried over (for example, FDDI, 802.5, ETHER, 802.3).
line1 Short string (less than 23 bytes) giving more information to be displayed on the same line as the source address. For example, "TYPE: 0x800" for Ethernet packets (NOT SNAP) or "LENGTH: 26" for 802.3 packets. You can leave this blank by passing "".
addlinfo Additional lines of information pertaining to data in the MAC header. (Blank for 802.3 and Ethernet, but could include formatted flags or other information in the MAC header for other link types). Must be terminated with a newline (\n).
upperinfo Other lines of information pertaining to data beyond the MAC header. Will be displayed only if the packet does not have 802.2 or Ethernet information present, that is, as in conjunction with set_up_link). Ordinarily must be left blank. If present, you may wish to include a separator.

DESCRIPTION

This routine formats the link level packet displaying all the upper layer information. This is the most detailed level of formatting and the subformatter must call this routine when nice formatting mode is enabled (the mode of formatting active is conveyed through ss_N_fmt_flag_type structure while calling subsys_N_format).

RETURN VALUES

0 Error in formatting
<> 0 Packet successfully formatted.

CONSTRAINTS

None
SEE ALSO

format_packet(9F), format_link_raw(9F), format_link_terse(9F), set_up_8022(9F), set_up_ether(9F),
set_up_link(9F), set_up_ip(9F), subsys_N_format(9F)
format_link_raw(9F)

NAME

format_link_raw - Formats the link level packet using raw formatting.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/subsys_id.h>

int format_link_raw(
    tl_msg_hdr_type *hdr,
    u_char *buffer,
    int len,
    int offset,
    char *linktype,
    char *interface,
    char *line3,
    char *addlinfo
);

PARAMETERS

hdr          Pointer to the standard nettl message header.
buffer       Pointer to entire traced packet (including MAC). Use the offset parameter to
             control where the data actually begins printing.
len           Length of entire buffer.
offset        Offset to actually begin displaying the data; that is, if the MAC information is
              not to be shown. 802.3 and Ethernet do not display until the beginning of the
              802.2 information or the Ethernet data (because the Source and Dest information
              are formatted out).
linktype      String describing the type of link this information is carried over, such as FDDI,
              802.3, 802.5, or Ethernet.
interface     String appended to the device ID and printed out in the
              interface= [interface] field. Pass a NULL pointer to suppress displaying
              any interface information.
line3         Short string, less than 14 bytes, giving information to be displayed on the same
              line as the addresses. For example, "Type: 0x800" for Ethernet packets (not SNAP)
              or "Length: 00-1a" for 802.3 packets. You can leave this blank by passing "".
addlinfo      (Blank for 802.3 and Ethernet, but may include formatted flags or other
              information in the MAC header for other link types). Terminated with a newline
              (\n).

DESCRIPTION

This routine formats the link level packet displaying all the upper layer information as hex/ASCII
data. The subformatter must call this routine when raw formatting mode is enabled (the mode
of formatting active is conveyed through ss_N_fmt_flag_type structure while calling
subsys_N_format).

RETURN VALUES

0      Error in formatting.
<> 0   Packet successfully formatted.
CONTRAINTS
None

SEE ALSO
format_packet(9F), format_link_nice(9F), format_link_terse(9F), set_up_8022(9F), set_up_ether(9F), set_up_link(9F), set_up_ip(9F), subsys_N_format(9F)
**format_link_terse(9F)**

**NAME**

format_link_terse – Formats the link level packet using terse formatting.

**SYNOPSIS**

```c
#include <fmt.h>
#include <ntl.h>
#include <sys/subsys_id.h>

int format_link_terse(
    tl_msg_hdr_type *hdr,
    u_char *buffer,
    int len,
    char *linktype,
    char *line3,
    char *addlinfo
);
```

**PARAMETERS**

- **hdr** Pointer to the standard nettl message header.
- **buffer** Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer.
- **len** Length of the buffer (including 802.2, excluding any lower layer data).
- **linktype** String describing the type of link this information is carried over, plus any other MAC layer information appropriate for terse mode. For 802.3 the linktype is simply "8"; for Ethernet (not SNAP) it is "E".
- **addlinfo** String giving other MAC or upper layer information to be displayed (blank for 802.3 and Ethernet).

**DESCRIPTION**

This routine formats the link level packet displaying the upper layer information in a single line. This is the least detailed level of formatting and the subformatter must call this routine when terse formatting mode is enabled (the mode of formatting active is conveyed through ss_N_fmt_flag_type structure while calling subsys_N_format).

**RETURN VALUES**

- 0 Error in formatting.
- <> 0 Packet successfully formatted.

**CONSTRAINTS**

None

**SEE ALSO**

format_packet(9F), format_link_nice(9F), format_link_raw(9F), set_up_8022(9F), set_up_ether(9F), set_up_link(9F), set_up_ip(9F), subsys_N_format(9F)
**NAME**

get_optParms_type - Structure containing subsystem options processing information.

**SYNOPSIS**

```c
#include<fmt.h>
```

**DESCRIPTION**

The `get_opt_parms_type` structure contains all the information required by the subformatter to process the options file. This structure is passed while calling the `subsys_N_get_options` function.

**Structure**

The `get_opt_parms_type` structure defined in `fmt.h` has the following fields:

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *</td>
<td>status_ptr</td>
</tr>
<tr>
<td>FILE *</td>
<td>subsys_strm</td>
</tr>
<tr>
<td>FILE *</td>
<td>error_strm</td>
</tr>
<tr>
<td>FILE *</td>
<td>log_strm</td>
</tr>
<tr>
<td>int</td>
<td>ss_id</td>
</tr>
<tr>
<td>char *</td>
<td>ss_name</td>
</tr>
<tr>
<td>nl_catd</td>
<td>ss_msg_cat</td>
</tr>
<tr>
<td>get_opt_flag_type</td>
<td>ss_n_get_opt_flag</td>
</tr>
<tr>
<td>char **</td>
<td>ss_options_ptr</td>
</tr>
<tr>
<td>int</td>
<td>ss_output_fd</td>
</tr>
<tr>
<td>char *</td>
<td>options_file_name</td>
</tr>
<tr>
<td>int *</td>
<td>options_filename_printed</td>
</tr>
</tbody>
</table>

`status_ptr` Contains the error code of the routine, if the returned value is -1. The `<fmt.h>` file gives a complete list of the 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` FILE 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 `subsys_N_options` routine, and each line has been converted to lower case. 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`  
FILE pointer to the file that will receive error messages.

`log_strm`  
FILE pointer to the file that will receive a summary of all options and files in effect for the subsystem, generated by `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`  
Subsystem ID number for the subsystem as found in the configuration file.

`ss_name`  
Subsystem name for the subsystem as found in the configuration file.

`ss_msg_cat`  
File descriptor pointing to the message catalog for the subsystem as found in the configuration file.

`ss_n_get_opt_flag`  
Type of flag is defined in `fmt.h` as:

```c
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 must not be used by `subsys_N_get_options`.

`parse_only_bit`  
This flag is set when the `subsys_N_get_options` routine does not need to process the information in the file, only parse the input and check for syntax and semantic errors.

`ss_options_ptr`  
Pointer to a pointer to a data structure containing the specific information processed by the `subsys_N_get_options` routine and passed on to the `subsys_N_format` routine to handle special formatting. This structure must be allocated and initialized by `subsys_N_get_options` routine.

`ss_output_fd`  
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` command with the `-c` option. The file name can be used in error and warning messages produced by `subsys_N_get_options` function while parsing the filter field. Subsequent messages need not display the file name.
options_filename_printed

Flag indicating if the file name needs to be printed in an error or warning message. A value of 0 indicates that the file name must be printed.

SEE ALSO

subsys_N_get_options(9F)
kget_log_instance(9F)

NAME
kget_log_instance - Produces a unique number for use as a log instance.

SYNOPSIS
#include <net_diag.h>
unsigned short kget_log_instance(void);

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 must 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 obtained earlier through a call to kget_log_instance.

RETURN VALUES
n  A unique number for use as a log instance value.

CONSTRAINTS
None

SEE ALSO
klogg_write(9F)
KLOG_CK(9F)

NAME

KLOG_CK - 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 subsystem ID of the calling subsystem (number assigned by
           Hewlett-Packard).

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 (given by subsys_id) and class (given by log_class).

RETURN VALUES

0  Logging is disabled.
1  Logging is enabled.

CONSTRAINTS

None

SEE ALSO

klogg_write(9F)
NAME

klogg_write - Sends log messages to the kernel trace and log facility.

SYNOPSIS

#include <sys/net_diag.h>
#include <sys/subsys_id.h>
int klogg_write(
    short subsys_id,
    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, enet driver's instance number) of the calling subsystem message. If this is a non-applicable parameter, pass in -1.

log_instance A unique static number used to identify the thread of events attending an interface. If this is a non-applicable parameter, pass in -1.

tl_packet Either a pointer to an mblk chain or a pointer to a set of iovec structures as determined by tl_packet_cnt. This structure is immediately copied into memory allocated by the tracing and logging facilities, so the calling routine need not copy the data and then pass a pointer to the data.

tl_packet_cnt If -2, tl_packet points to an mblk 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

None

SEE ALSO

KLOG_CK(9F)
KTRC_CHECK(9F)

NAME

KTRC_CHECK - Checks whether tracing is enabled; verifies tracing activation on a per-interface basis.

SYNOPSIS

#include <sys/net_diag.h>
#include <sys/subsys_id.h>

int KTRC_CHECK(
    int subsys_id,
    int trace_kind,
    int device_id
);

PARAMETERS

subsys_id The unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard).

trace_kind The kind of trace. Available kinds are defined in the <sys/subsys_id.h> header file as follows:
    ERROR_TRACE_BIT     Error tracing mask.
    HDR_IN_BIT          Inbound header tracing mask.
    HDR_OUT_BIT         Outbound header tracing mask.
    LOGGING_TRACE_BIT   Log call tracing mask.
    LOOP_BACK_BIT       For loopback.
    PDU_IN_BIT          Inbound PDU tracing mask.
    PDU_OUT_BIT         Outbound PDU tracing mask.
    PROCEDURE_TRACE_BIT Procedure entry/exit trace.
    PTO/P_BIT           For point to point.

device_id The device ID number (for example, enet driver’s instance number). 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 (given by subsys_id), kind (given by trace_kind), and interface (given by device_id).

In a system with more than one interface card installed, this macro allows tracing on a per-interface basis, thereby reducing the impact of tracing on performance.

RETURN VALUES

0   Tracing is disabled.
1   Tracing is enabled.

CONSTRAINTS

None

EXAMPLES

The enet driver might use this macro as follows to check if tracing is enabled on the current instance for outbound data.
if (KTRC_CHECK(ENET_ID, PDU_OUT_BIT,
            (enet_iftp->lancift.hwift.instance_num))
    {
        ktrc_write(...);
    }

SEE ALSO

ktrc_write(9F), KTRC_CK(9F)
KTRC_CK(9F)

NAME

KTRC_CK - Checks whether tracing is enabled.

SYNOPSIS

#include <sys/net_diag.h>
#include <sys/subsys_id.h>

int KTRC_CK(
    int subsys_id,
    int trace_kind
);

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.
    PTOB_BIT For point to point.

DESCRIPTION

The KTRC_CK network service is a macro that verifies whether tracing is enabled for the current
subsystem (given by subsys_id) and kind (given by trace_kind).

The difference between this macro and KTRC_CHECK macro is that KTRC_CK does not include
the per-interface check.

RETURN VALUES

0 Tracing is disabled.
1 Tracing is enabled.

CONSTRAINTS

None

EXAMPLES

The enet driver uses this macro as follows to check if tracing is enabled for outbound data:

    if (KTRC_CK(ENET_ID, PDU_OUT_BIT)
        {
            ktrc_write(...);
        }
SEE ALSO

ktrc_write(9F), KTRC_CHECK(9F)
NAME

ktrc_write - Sends trace messages to kernel trace and log facility.

SYNOPSIS

#include <sys/net_diag.h>
#include <sys/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 non-applicable parameter, pass in -1.

device_id The device ID number (for example, enet driver’s instance number) of the calling subsystem message. If this is a nonapplicable parameter, pass in -1.

tl_packet Either a pointer to an mblk 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 mblk chain or an iovec structure. This structure is immediately copied into a memory location 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 -2, tl_packet points to an mblk chain. If greater than 0, this is the number of the iovec structure to which tl_packet points.

RETURN VALUES
Always returns a 0.

CONSTRAINTS
None

SEE ALSO
KTRC_CHECK(9F), KTRC_CK(9F)
NAME
set_up_8022 - Sets up global information for the filter and formatting functions.

SYNOPSIS
#include <fmt.h>
#include <nt1.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 function sets up global information used by the filter_packet,
format_link_nice, format_link_raw, and format_link_terse filter and formatting
functions. 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 must not include MAC
information.

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
None

SEE ALSO
set_up_ether(9F), set_up_ip(9F), set_up_link(9F)
set_up_ether(9F)

NAME

set_up_ether - Sets up global information for the filter and formatting functions.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <subsys_id.h>

int set_up_ether(
    u_char *buf_ptr,
    int len,
    u_char *dst_addr,
    u_char *src_addr,
    int ether_type
);

DESCRIPTION

The set_up_ether function sets up global information used by the filter_packet, format_link_nice, format_link_raw, and format_link_terse filter and formatting functions. 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 must 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

None

SEE ALSO

set_up_8022(9F), set_up_ip(9F), set_up_link(9F)
NAME

set_up_ip - Sets up global information for the filter and formatting functions.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/subsys_id.h>

int set_up_ip(
    u_char *buf_ptr,
    int len
);

DESCRIPTION

The set_up_ip function sets up global information used by the filter_packet, format_link_nice, format_link_raw, and format_link_terse filter and formatting functions. 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 must not use this routine. Call this routine only when no link information is available for formatting output (for example, NS_LOOPBACK).

PARAMETERS

buf_ptr Pointer to the beginning of the IP information, which must not include MAC information.

len Length of the buffer, excluding the MAC header.

RETURN VALUES

Always returns 0.

CONSTRAINTS

None

SEE ALSO

set_up_8022(9F), set_up_ether(9F), set_up_link(9F)
set_up_link(9F)

NAME

set_up_link - Sets up global information for the link layer only.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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 must not include MAC information. 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 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

None

SEE ALSO

set_up_8022(9F), set_up_ether(9F), set_up_ip(9F)
ss_N_fmt_flag_type(9S)

NAME

ss_N_fmt_flag_type - Flags structure that control the behavior of formatter output.

SYNOPSIS

#include<fmt.h>

DESCRIPTION

The ss_N_fmt_flag_type structure contains flags that dictate the desired format of trace and log data output. This structure is exchanged between the core formatter and subformatter through various functions, such as tl_header_format1 and subsys_N_format.

Structure

The ss_N_fmt_flag_type structure has the following field names:

Table 4-2 Structure Fields (ss_N_fmt_flag_type)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>verbosity_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>console_logging</td>
<td>unsigned</td>
</tr>
<tr>
<td>highlight_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>nice_mode_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>terse_mode_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>terse_link_mode_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>terse_time_mode_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>map_to_names_bit</td>
<td>unsigned</td>
</tr>
<tr>
<td>reserved</td>
<td>unsigned</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>verbosity_bit</td>
<td>When this bit is set, a high level of verbosity has been selected (high verbosity is the default).</td>
</tr>
<tr>
<td>console_logging</td>
<td>This bit is set if console logging is enabled, in which case the subformatter must only call the tl_header_format1 routine and provide very minimal additional information (to be kept to one line).</td>
</tr>
<tr>
<td>highlight_bit</td>
<td>This bit is set when nice formatting has been enabled (nice output not enabled is the default). Nice formatting is the most descriptive mode of formatting. All possible information must be displayed in this mode of output. Nice mode is not usually used for log messages.</td>
</tr>
<tr>
<td>terse_mode_bit</td>
<td>This bit is set when terse formatting has been enabled (terse output not enabled is the default). Terse formatting must be limited to one line of output per trace record. Terse mode is not usually used for log messages.</td>
</tr>
<tr>
<td>terse_link_mode_bit</td>
<td>If terse mode is enabled then terse_link_mode_bit is a flag that must cause the link name to be included in the output.</td>
</tr>
<tr>
<td>terse_time_mode_bit</td>
<td>If terse mode is enabled, terse_time_mode_bit is a flag that must cause the timestamp to be included in the output.</td>
</tr>
</tbody>
</table>
map_to_names_bit  This bit is set when numbers must be resolved into names whenever possible (mapping numbers to names is enabled by default). For example, an IP address must be displayed as a host name if the map_to_names_bit flag is set.

reserved  Reserved

SEE ALSO

tl_header_format1(9F), subsys_N_form(9F)
ss_N_fmt_Parms_type(9S)

NAME

ss_N_fmt_Parms_type - Structure containing information required by subformatter.

SYNOPSIS

#include<fmt.h>

DESCRIPTION

The ss_N_fmt_Parms_type structure contains all the information required by a subformatter to operate. The subformatter can obtain this information by calling the tl_get_parms function. The core formatter builds and initializes this data structure before calling subsys_N_format.

Structure

The ss_N_fmt_Parms_type structure has the following field names:

Table 4-3 Structure Fields (ss_N_fmt_Parms_type)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *</td>
<td>ss_status_ptr</td>
</tr>
<tr>
<td>FILE *</td>
<td>ss_output_strm</td>
</tr>
<tr>
<td>int</td>
<td>ss_output_fd</td>
</tr>
<tr>
<td>FILE *</td>
<td>ss_error_strm</td>
</tr>
<tr>
<td>int</td>
<td>ss_error_fd</td>
</tr>
<tr>
<td>nl_catd</td>
<td>ss_msg_catd</td>
</tr>
<tr>
<td>char *</td>
<td>ss_name</td>
</tr>
<tr>
<td>char *</td>
<td>ss_binary_msg_ptr</td>
</tr>
<tr>
<td>char *</td>
<td>ss_options_ptr</td>
</tr>
<tr>
<td>ss_N_fmt_flag_type</td>
<td>ss_n_fmt_flag</td>
</tr>
<tr>
<td>char *</td>
<td>time_buffer</td>
</tr>
<tr>
<td>int</td>
<td>time_buffer_length</td>
</tr>
<tr>
<td>int</td>
<td>output_file_count</td>
</tr>
<tr>
<td>fd_result</td>
<td>output_files[1]</td>
</tr>
<tr>
<td>int</td>
<td>print_op</td>
</tr>
<tr>
<td>int</td>
<td>user_count</td>
</tr>
<tr>
<td>user_acct_result *</td>
<td>users</td>
</tr>
<tr>
<td>int</td>
<td>inited_flag</td>
</tr>
<tr>
<td>int</td>
<td>nettl_version</td>
</tr>
</tbody>
</table>

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 subsys_N_format() (9F).
ss_n_fmt_flags
  Options flags of type ss_N_fmt_flag_type, which indicates the
  desired format of output.
time_buffer
  A string containing the formatted timestamp from the trace/log
  header (see the time-buffer parameter of subsys_N_format() (9F)).
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.
inited_flag
  Flag which indicates if the structure is already (initialized or not.)
netttl_version
  Gives the current netttl version.

SEE ALSO
catgets(9F), tl_get_parms(9F), ss_N_fmt_flag_type(9S), subsys_N_format(9F), subsys_N_get_options(9F)
NAME

subsys_N_format - Routine to format a single trace or log message from subsystem N.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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. Pass the name to Network Services by including it in a shared library that is specified 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 must 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 must 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 must be masked. The subformatter must never call exit. The definitions of the various flags must be carefully followed. Subsystems must try to mirror the behavior of other subsystems as much as possible.

NOTE: Do not 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 \texttt{tl\_get\_parms} function anywhere within the subformatter to get a pointer to all of the information that a subformatter might need. Use this \texttt{tl\_get\_parms} mechanism whenever possible, as explained further in \texttt{tl\_get\_parms}(9F).

All future parameter changes will be made through the \texttt{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 \texttt{tl\_get\_parms}.

**PARAMETERS**

\textit{flags} \hfill Flags of type \texttt{ss\_N\_fmc\_flag\_type} that indicates the desired output format.

\textit{binary-msg-ptr} \hfill A pointer to a buffer that contains the binary trace or log message to be formatted. The buffer contains the trace/log header, a \texttt{tl\_msg\_hdr\_type} structure (from the \texttt{<ntl.h>} file), followed by the trace/log data (from \texttt{ktrc\_write} or \texttt{klogg\_write}):

\begin{verbatim}
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;
\end{verbatim}

**NOTE:** For tracing, the data may be truncated by the nettl command facilities. Check the \texttt{tl\_msg\_hdr\_type\_data\_len} field to find out how much data was captured.

\textit{options-ptr} \hfill A pointer to a data structure defined by the subsystem for communication between the \texttt{subsys\_N\_get\_options} routine and the \texttt{subsys\_N\_format} routine. If no options are used, this pointer is \texttt{NULL}. The actual type of the structure pointed to by \texttt{options-ptr} is entirely up to the subsystem developer.

\textit{msg-cat-fd} \hfill The file descriptor of the subsystem message catalog configured in \texttt{nettlgen.conf}. The formatter opens subsystem message catalogs using \texttt{catopen}. The \texttt{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 must not open their own message catalog files.

\textit{error-fd} \hfill A file descriptor that identifies the file that receives any fatal or nonfatal error messages (typically associated with \texttt{stderr}).

\textit{output-file-count} \hfill 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, \texttt{fd}, and a result.

```c
typedef struct {
    int fd;
    int result;
} fp_result;
```

The \texttt{fd}-designated output file, \texttt{output-files[0].fd}, receives the formatted trace/log messages. Only one output file is used for HP-UX; \texttt{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:

- \texttt{FMTERR_INV_FLAGS} Invalid flags parameter.
- \texttt{FMTERR_INV_BIN_MP} Invalid binary-msg-ptr binary message pointer.
- \texttt{FMTERR_INV_OUT_FP} Invalid output file pointer.
- \texttt{FMTERR_INV_MC_FP} Invalid message catalog file pointer.
- \texttt{FMTERR_INV_TL_MSG} Invalid trace/log message. The message is so corrupted that no formatting can be done.
- \texttt{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 \texttt{error-fd}.

CONSTRAINTS

None

SEE ALSO

catopen(9F), exit(9F), klogg_write(9F), ktrc_write(9F), subsys_N_get_options(9F),
tl_check_cat_version(9F), tl_format_fprintf(9F), tl_format_write(9F), tl_get_parms(9F),
tl_header_format1(9F), tl_raw_format(9F)
NAME

subsys_N_get_options - Routine to process options for subsystem N.

SYNOPSIS

#include <fmt.h>

int subsys_N_get_options(
    get_opt_parms_type *get-opt-parms-ptr
);

DESCRIPTION

The subsys_N_get_options network function is provided by the subsystem developer. It can have any unique name. Pass the name to Network Services by including it in a shared library that is specified in the nettlgen.conf configuration file. With this naming notification method, several subsystems can use the same subsys_N_get_options function. Note that if the subsystem provides both subsys_N_format and subsys_N_get_options functions, they must be contained in the same shared library specified in the nettlgen.conf configuration 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 (given by subsys_strm). 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. 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. See get_opt_parms_type(9F) for explanation of the structure fields.

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

None

SEE ALSO

get_opt_parms_type(9F), malloc(3C), nettl(1M), subsys_N_format(9F), tl_get_line(9F)
NAME

tl_banner_char - Gets the character used for tl_header_format1 banner printing.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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 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 and log messages.

RETURN VALUES

The tl_banner_char always returns a character to be used by tl_header_format1.

CONSTRAINTS

None

SEE ALSO

tl_header_format1(9F)
The tl_check_cat_version function checks that the subsystem message catalog has a compatible version with the subsystem formatter library. If not, an appropriate warning message is issued.

This function is (optionally) called by the subsys_N_format and subsys_N_get_options functions.

RETURN VALUES

0  Successful completion. The versions match.
-1  Error. The versions do not match or the file descriptor of the message catalog is invalid.

CONSTRAINTS

None

SEE ALSO

tl_header_format1(9F) subsys_N_format(9F), subsys_N_get_options(9F)
NAME

tl_format_fprintf - Converts, format, and print arguments to standard output.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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 arguments for the format. If the format is exhausted while arguments remain, the excess arguments 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
None

SEE ALSO
subsys_N_format(9F), tl_format_write(9F), tl_get_parms(9F), tl_raw_format(9F)
tl_format_write(9F)

NAME

tl_format_write - Writes a buffer to standard output.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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,
    fd_result output_files[],
    int print_op,
    int user_count,
    user_acct_result users[],
    error_num *status_ptr
);

PARAMETERS

input_line_ptr A character string that contains the message to be printed to
the output files. The input_line_ptr parameter 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 Reserved for future use.

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**

None

**SEE ALSO**

`tl_format_fprint(9F), tl_raw_format(9F)`
NAME
tl_get_line - Obtains a line from a filter command file.

SYNOPSIS
int tl_get_line(
    FILE *commandstream,
    char *line,
    int linesize,
    char *origline,
    int lineno,
    FILE *errorstream
);

PARAMETERS
commandstream  A FILE pointer that points to the temporary filter command file containing
                a single subsystem's filter commands; typically the subsys_strm field of
                the get_opt parms_type parameter to subsys_N_get_options.
line            The buffer where tl_get_line stores the filter command line.
linesize        Size of line (no more than 2048).
origline        The tl_get_line routine stores the original filter command line as it
                appeared in the filter command file in origline.
lineno          The tl_get_line routine stores 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. The tl_get_line
function stores the "cleaned" filter command line in this buffer.

RETURN VALUES
2    Successful completion.
0    End of file.
<0   Error.

CONSTRAINTS
None

SEE ALSO
subsys_N_get_options(9F)
**NAME**

tl_get_parms - Returns a pointer to an ss_N_fmt_parms_type data structure.

**SYNOPSIS**

```c
#include <fmt.h>
#include <ntl.h>
#include <sys/subsys_id.h>

ss_N_fmt_parms_type *tl_get_parms(void);
```

**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`. See `ss_N_fmt_parms_type(9S)` for description of this structure.

**RETURN VALUES**

- 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**

None

**SEE ALSO**

`subsys_N_format(9F)`
tl_header_format1(9F)

NAME

tl_header_format1 - Routine to format a single trace or log header.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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,
    error_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 ss_N_fmt_flag_type(9S).

kind_str A text message (typically the result of the tl_log_class or
tl_trace_kind function) to be displayed for the kind field from
the trace/log header. This string must be null-terminated. The kind
message is truncated to 16 characters. If kind_str is NULL, the kind
field from the header is displayed as a decimal value.

banner_char The character to use in the banner header line (typically the result
of the tl_banner_char function). The subformatter may use this
character to indicate differences in messages, such as inbound or
outbound messages. For example, inbound messages could use the
character "v" while outbound messages could use the character "^".

output_file_count This is the number of output files to receive the formatted trace/log
header output. For HP-UX, only one output file is used; this value
is always 1.

output_files[] An array of structures consisting of a file descriptor and a result
variable for each file to receive the formatted trace/log header
output. For HP-UX, only one output file is used:
output_file[0].fd.
*time_buffer*  
A string depicting the formatted time stamp from the trace/log header.

*time_buffer_length*  
The length of *time_buffer* not counting the null terminator byte.

*print_op*  
For HP-UX, this parameter must have a value of 0.

*user_count*  
For HP-UX, this parameter must have a value of 0.

*users*  
For HP-UX, this parameter must have a value of NULL.

*location*  
A value that can be used to locate the source of the message in the code. This parameter is set by the subsystem and may be used to represent any information the subsystem desires.

*status_ptr*  
The error value if the routine returns a -1:

- **FMTERR_INV_HDR**  
The trace/log header is invalid (corrupt).

- **FMTERR_INV_HDR_PTR**  
The trace/log header pointer is invalid.

- **FMTERR_INV_OUT_FD**  
The output file descriptor is invalid.

- **FMTERR_INV_MC_FD**  
The message catalog descriptor is invalid.

- **FMTERR_SYS_ERROR**  
An error was returned from a system call within *tl_header_format1*.

**DESCRIPTION**

The *tl_header_format1* network function formats a single trace or log header. The format of the output conforms to the standard HP-UX network tracing and logging recommendations. The formatted header is written to the output file specified by `output_file[0]. fd`. The *tl_header_format1* function 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**

None

**SEE ALSO**

`tl_banner_char(9F), tl_get_parms(9F), tl_log_class(9F), tl_trace_kind(9F)`
tl_log_class(9F)

NAME

tl_log_class - Returns 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>.

class   Keyword
1       INFORMATIVE
2       WARNING
4       ERROR
8       DISASTER

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

None

SEE ALSO

tl_trace_kind(9F)
NAME

tl_raw_format - Formats trace or log message into hexadecimal and printable ASCII characters.

SYNOPSIS

#include <fmt.h>
#include <ntl.h>
#include <sys/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[],
    error_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 is to begin. If start is zero, the dump begins at the byte pointed to by data_ptr.

error_fd A file descriptor that will receive error messages.

flags Reserved for future used; must 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:
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**

None

**SEE ALSO**

`tl_format_fprintf(9F), tl_format_write(9F)`
NAME

tl_trace_kind - Returns 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.

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

None

SEE ALSO

tl_log_class(9F)
SCSI Services is a set of commonly used SCSI functions that allow device and interface drivers to be much smaller and more supportable. In addition to providing most commonly used SCSI functions, WSIO SCSI Services also provides a supported pass-through mechanism.
dd_close(9E)

NAME

dd_close - 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(9F) and minor(9F)).

DESCRIPTION

The dd_close SCSI function is provided by the driver writer. It can have any unique name. Pass the dd_close function pointer to SCSI Services by specifying it in the dd_close field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

None

CONSTRAINTS

None

SEE ALSO

scsi_lun_close(9F), scsi_ddsw(9S)
dd_done(9E)

NAME
dd_done - SCSI driver entry point to handle post-I/O processing.

SYNOPSIS

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 dd_done function pointer to SCSI Services by specifying it in the dd_done field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

The dd_done is declared as returning int; however, its return value is not used by SCSI services.

CONSTRAINTS

None

SEE ALSO

biodone(9F), scsi_action(9F), scsi_ddsw(9S)
**dd_ioctl(9E)**

**NAME**

dd_ioctl - SCSI driver entry point to handle device I/O controls.

**SYNOPSIS**

```c
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 `dd_ioctl` function pointer to SCSI Services by specifying it in the `dd_ioctl` field of the `scsi_ddsw` structure.

See the *HP-UX 11i v2 Driver Development Guide* for information.

**RETURN VALUES**

- `0`   Successful completion.
- `!=0` Error. Return an errno value.

**CONSTRAINTS**

None

**SEE ALSO**

`scci_cmd(9F), scsi_init_inquiry_data(9F), scsi_ioctl(9F)`
**NAME**

`dd_ioctl_okay` - SCSI driver entry point to allow/disallow ioctl commands sent through the pass-through driver.

**SYNOPSIS**

```c
int dd_ioctl_okay(
    dev_t dev,
    int  cmd,
    caddr_t data,
    int  flags
);
```

**PARAMETERS**

- **dev**: Device number
- **cmd**: Command
- **data**: Pointer to command parameters
- **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 `dd_ioctl_okay` function pointer to SCSI Services by specifying it in the `dd_ioctl_okay` field of the `scsi_ddsw` structure.

See the HP-UX 11i v2 Driver Development Guide for information.

**RETURN VALUES**

- **PT_OKAY**: Successful completion.
- **0**: Error

**CONSTRAINTS**

None

**SEE ALSO**

`scsi_ioctl(9F)`
dd_open(9E)

NAME

dd_open - SCSI driver entry point to handle device open.

SYNOPSIS

int 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 dd_open function pointer to SCSI Services by specifying it in the dd_open field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0 Successful completion.
!=0 Error. Return an errno value.

CONSTRAINTS

None

SEE ALSO

m_scsi_lun(9F), major(9F), scsi_cmdx(9F), scsi_init_inquiry_data(9F), scsi_lun_open(9F)
dd_pass_thru_done(9E)

NAME
dd_pass_thru_done - SCSI driver entry point to handle post-pass-through I/O processing.

SYNOPSIS

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 dd_pass_thru_done function pointer to SCSI Services by specifying it in the dd_pass_thru_done field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

The dd_pass_thru_done is declared as returning int; however, the return value is not used by SCSI services.

CONSTRAINTS

None
dd_pass_thru_okay(9E)

NAME

dd_pass_thru_okay - SCSI driver entry point to control pass-through I/O requests.

SYNOPSIS

int dd_pass_thru_okay(
    dev_t dev,
    struct sctl_io *sctl_io
);

PARAMETERS

dev Device number.
sctl_io Structure 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 dd_pass_thru_okay function pointer to SCSI Services by specifying it in the
dd_pass_thru_okay field of the scsi_ddsw structure.
See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

PT_OKAY Successful completion.
0 Error.

CONSTRAINTS

None
dd_read(9E)

NAME
dd_read - SCSI driver entry point to handle device read operations.

SYNOPSIS

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 dd_read function pointer to SCSI Services by specifying it in the dd_read field of the
scsi_ddsw structure.
See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0   Successful completion.
!=0 Error. Return an errno value.

CONSTRAINTS

None

SEE ALSO

scsi_read(9F)
NAME

dd_start - 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 (scb_structure).

DESCRIPTION

The dd_start SCSI function is provided by the driver writer. It can have any unique name. Pass the dd_start function pointer to SCSI Services by specifying it in the dd_start field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

<>NULL Successful completion. A pointer to a valid buf structure is returned.

NULL Error.

CONSTRAINTS

None
dd_strategy(9E)

NAME

dd_strategy - SCSI driver entry point to handle buf requests.

SYNOPSIS

int dd_strategy(
    struct buf *bp,
    struct scsi_lun *lp
);

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 dd_strategy function pointer to SCSI Services by specifying it in the dd_strategy field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0    Successful completion.
-1    Error.

CONSTRAINTS

The dd_strategy function must exist (be defined as non-NULL in the scsi_ddsw structure) if your driver calls scsi_strategy. The dd_strategy function shall not try to acquire SCSI lun lock as scsi_strategy calls dd_strategy while holding the lun lock.

SEE ALSO

physio(9F), dd_read(9E), dd_write(9E), scsi_enqueue(9F), scsi_strategy(9F)
NAME

dd_write - 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 dd_write function pointer to SCSI Services by specifying it in the dd_write field of the scsi_ddsw structure.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0 Successful completion.
!=0 Error. Replace an errno value.

CONSTRAINTS

None

SEE ALSO

physio(9F), scsi_write(9F)
NAME

driver_if_abort - Interface driver specific SCSI abort function.

SYNOPSIS

int driver_if_abort(
    struct buf *bp
);

PARAMETERS

bp Pointer to a buf structure

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 driver_if_abort function to SCSI services by specifying it in the if_abort field of the scsi_ifsw structure. Commonly, driver is replaced by your driver's name.

The driver_if_abort function provides a means 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 SIOC_ABORT ioctl request.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0 Success.
!=0 Error. Return an errno value.

CONSTRAINTS

None

SEE ALSO

scsi_ifsw(9F)
driver_if_bdr(9F)

NAME

driver_if_bdr - Interface driver specific SCSI Bus Device Reset function.

SYNOPSIS

int driver_if_bdr(
    dev_t dev
);

PARAMETERS

dev  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 driver_if_bdr function pointer to SCSI services by specifying it in the if_bdr field of the scsi_ifsw structure. Commonly, driver is replaced by your driver's name.

The driver_if_bdr function provides a means 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 11i v2 Driver Development Guide for information.

RETURN VALUES

0   Success.

!=0   Error. Return an errno value.

CONSTRAINTS

None

SEE ALSO

scsi_ifsw(9F)
**NAME**

driver_if_close - Interface driver specific logical unit close processing.

**SYNOPSIS**

```c
int driver_if_close(
    dev_t dev
);
```

**PARAMETERS**

- `dev`  
  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. Pass the `driver_if_close` function pointer to SCSI services by specifying it in the `if_close` field of the `scsi_ifsw` structure. Commonly, `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 the `driver_if_close` function with the device number of the device being opened as its sole argument.

See the *HP-UX 11i v2 Driver Development Guide* for information.

**RETURN VALUES**

- `0`  Success.
- `!0`  Error. Return an `errno` value.

**CONSTRAINTS**

The `driver_if_close` function 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(9F)`
NAME

driver_if_open - Interface driver specific logical unit open processing.

SYNOPSIS

int driver_if_open(
    dev_t dev
);

PARAMETERS

dev 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. Pass the driver_if_open function pointer to
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 the driver_if_open
function with the device number of the device being opened as its sole argument.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0 Success.
!=0 Error. Return an errno value.

CONSTRAINTS

The driver_if_open function 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 the driver_if_open function returns.

SEE ALSO

scsi_ifsw(9F)
driver_if_reset_bus(9F)

NAME

driver_if_reset_bus - Interface driver specific SCSI bus reset function.

SYNOPSIS

int driver_if_reset_bus(
    dev_t dev
);

PARAMETERS

dev Device number

DESCRIPTION

The SCSI subsystem allows, but does not require, the interface driver to specify a bus reset function. It can have any unique name. Pass the driver_if_reset_bus function pointer to SCSI services by specifying it in the if_reset_bus field of the scsi_ifsw structure. Commonly, driver is replaced by your driver's name.

When the SCSI subsystem wants to reset a bus, it checks the if_reset_bus field of the scsi_ifsw structure for the bus. If the if_reset_bus is not NULL, the driver_if_reset_bus function is called with a device number identifying the bus as its sole argument.

Any outstanding I/Os on the bus at the time of the reset are returned to the SCSI subsystem with the appropriate status field set to SCTL_INCOMPLETE. That is, if it was the Request Sense resulting from a check condition that was terminated by the reset, then scb->sense_action is set to SCTL_INCOMPLETE. Otherwise, scb->cdb_status is set to SCTL_INCOMPLETE. “struct scb” is described under data structures later in this section.

The SCSI subsystem makes this call only in response to SIOC_RESET_BUS ioctl request. See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

0 Success.

!=0 Error. Return an errno value.

CONSTRAINTS

None

SEE ALSO

scsi_ifsw(9F)
driver_if_start(9F)

NAME
driver_if_start - Interface driver specific start function.

SYNOPSIS

void driver_if_start(
   struct isc_table_type *isc
);

PARAMETERS

isc A pointer to a isc_table_type structure.

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.

The driver_if_start function can have any unique name. Pass the driver_if_start
function pointer to 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 called with a pointer
to the isc_table_type structure as its sole argument.

The SCSI subsystem may call if_start at any time. For example, 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 completes the I/O.

See the HP-UX 11i v2 Driver Development Guide for information.

RETURN VALUES

None

CONSTRAINTS

The driver_if_start is not permitted to sleep under any circumstances.

SEE ALSO

scsi_ifsw(9F)
m_bus_id(9F)

NAME
m_bus_id - Extracts SCSI Bus instance number.

SYNOPSIS
#include <sys/scsi_ctl.h>

m_bus_id(
    dev_t dev
);

PARAMETERS
dev    Device number

DESCRIPTION
The m_bus_id macro extracts the bus instance number of the SCSI bus corresponding to dev.

RETURN VALUES
The m_bus_id macro does not return any values.

CONSTRAINTS
None

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 instance */
    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
m_lun_id(9F)

NAME
m_lun_id - Extracts SCSI LUN ID.

SYNOPSIS
#include <sys/scsi_ctl.h>

m_lun_id(
    dev_t dev
);

PARAMETERS

dev     Device number

DESCRIPTION
The m_lun_id macro extracts the ID of the SCSI LUN corresponding to dev.

RETURN VALUES
The m_lun_id macro does not return any values.

CONSTRAINTS
None

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 instance */
    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
m_scsi_bus(9F)

NAME
m_scsi_bus - SCSI bus pointer.

SYNOPSIS
#include <sys/scsi_ctl.h>
struct scsi_bus *m_scsi_bus(
    dev_t dev
);

PARAMETERS
dev Device number

DESCRIPTION
The m_scsi_bus function returns a pointer to the scsi_bus structure corresponding to dev.

RETURN VALUES
NULL Error.
<>0 Pointer to the SCSI bus structure associated with dev.

CONSTRAINTS
None

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);
    if(busp == NULL) {
        msg_printf("mydriver - a NULL scsi_bus
        pointer\n");
        return (ENXIO);
    }

    /* Get the pointer to scsi_tgt structure */
    tgtp = m_scsi_tgt(dev);

    /* Get the pointer to scsi_lun structure */
    lunp = m_scsi_lun(dev);
    . . .
}
SEE ALSO
NAME
m_scsi_isc - Returns isc_table_type pointer.

SYNOPSIS
#include <sys/scsi_ctl.h>
struct isc_table_type *m_scsi_isc(
    dev_t dev
);

PARAMETERS
  dev Device number

DESCRIPTION
The m_scsi_isc function returns a pointer to the isc_table_type structure corresponding to dev.

RETURN VALUES
NULL  Error.
<>0  Pointer to the ISC structure associated with dev.

CONSTRAINTS
None

EXAMPLES
#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
m_scsi_lun(9F)

NAME

m_scsi_lun - Returns scsi_lun pointer.

SYNOPSIS

#include <sys/scsi_ctl.h>

struct scsi_lun *m_scsi_lun(
    dev_t dev
);

PARAMETERS

dev Device number

DESCRIPTION

The m_scsi_lun function returns a pointer to the scsi_lun structure corresponding to dev.

RETURN VALUES

NULL Error.
<>0 Pointer to the SCSI LUN structure associated with dev.

CONSTRAINTS

None

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);
    if(busp == NULL) {
        msg_printf("mydriver - a NULL scsi_bus
            pointer\n");
        return (ENXIO);
    }

    /* Get the pointer to scsi_tgt structure */
    tgtp = m_scsi_tgt(dev);

    /* Get the pointer to scsi_lun structure */
    lunp = m_scsi_lun(dev);
    ....
}
SEE ALSO
**NAME**

m_scsi_tgt - Returns scsi_tgt pointer.

**SYNOPSIS**

```c
#include <sys/scsi_ctl.h>
struct scsi_lun *m_scsi_tgt(
    dev_t dev
);
```

**PARAMETERS**

*dev*    Device number

**DESCRIPTION**

The `m_scsi_tgt` function returns a pointer to the `scsi_tgt` structure corresponding to `dev`.

**RETURN VALUES**

- NULL   Error.
- <>0   Pointer to the SCSI target structure associated with `dev`.

**CONSTRAINTS**

None

**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);
    if(busp == NULL) {
        msg_printf("mydriver - a NULL scsi_bus pointer\n");
        return (ENXIO);
    }

    /* Get the pointer to scsi_tgt structure */
    tgtp = m_scsi_tgt(dev);

    /* Get the pointer to scsi_lun structure */
    lunp = m_scsi_lun(dev);
    ....
}
```
m_tgt_id(9F)

NAME
m_tgt_id - Extracts SCSI target ID.

SYNOPSIS
#include <sys/scsi_ctl.h>
m_tgt_id(
    dev_t  dev
);

PARAMETERS
dev  Device number

DESCRIPTION
The m_tgt_id macro extracts the ID of the SCSI target corresponding to dev.

RETURN VALUES
The m_tgt_id macro does not return any values.

CONSTRAINTS
None

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 instance */
    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

scb - SCSI Control Block Structure.

SYNOPSIS

#include <sys/scsi_ctl.h>

DESCRIPTION

SCSI services allocate scb structure and associate it with a buf structure. The fields in the scb structure hold temporary state information until an I/O is completed.

The SCB 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 has fields to pass any sense data that an interface driver may return 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 table shows a list of driver accessible fields:

Table 5-1 Relevant scb Structure Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_scb</td>
<td>void*</td>
</tr>
<tr>
<td>lp</td>
<td>struct scsi_lun *</td>
</tr>
<tr>
<td>flags</td>
<td>ubit32</td>
</tr>
<tr>
<td>max_msecs</td>
<td>ubit32</td>
</tr>
<tr>
<td>cdb</td>
<td>ubit8</td>
</tr>
<tr>
<td>cdb_len</td>
<td>ubit8</td>
</tr>
<tr>
<td>io_id</td>
<td>ubit32</td>
</tr>
<tr>
<td>tag</td>
<td>ubit8</td>
</tr>
<tr>
<td>cdb_status</td>
<td>ubit32</td>
</tr>
<tr>
<td>data_resid</td>
<td>ubit32</td>
</tr>
<tr>
<td>sense_status</td>
<td>ubit32</td>
</tr>
<tr>
<td>sense_bytes</td>
<td>ubit8</td>
</tr>
<tr>
<td>sense_data</td>
<td>ubit8*</td>
</tr>
</tbody>
</table>

if_scb 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’d 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 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 structures as opposed to a per-bus pool, 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 the following bits in the flag for proper functionality:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCB_2BYTE</td>
<td>Is the same as SCB_4BYTE except that phase changes are only restricted to even boundaries.</td>
</tr>
<tr>
<td>SCB_4BYTE</td>
<td>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-&gt;b_bcount. Note that SCB_4BYTE does not imply that bp-&gt;b_count is a multiple of four or that bp-&gt;b_un.b_addr is four-byte aligned. Note also that the phase change out of data phase if all bp-&gt;b_bcount bytes have been transferred is not subject to the alignment restructuring.</td>
</tr>
<tr>
<td>SCB_NO_DISC</td>
<td>Indicates not to grant the disconnect privilege in the identify message.</td>
</tr>
<tr>
<td>SCB_ORDERED_TAG</td>
<td>Denotes that ordered tags are intended to be used for this device.</td>
</tr>
<tr>
<td>SCB_SDTR</td>
<td>If this bit is set and SCB_WDTR is not set, the interface driver must initiate SDTR negotiation immediately following the Selection, Identify, or tag message, whichever comes last, and before sending the CDB for the I/O.</td>
</tr>
<tr>
<td>SCB_WDTR</td>
<td>Directs the interface driver to initiate a wide negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending any CDB for the I/O. If (tp-&gt;state and T_ENABLE_SDTR) or (scb-&gt;flags and SCB_SDTR) is also set, the interface driver must initiate SDTR negotiation immediately following the WDTR negotiation. The wide negotiation must always precede the synchronous negotiation because a wide negotiation resets the link to asynchronous negotiation.</td>
</tr>
</tbody>
</table>

### 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 must never abort this I/O based solely on the amount of time since Selection.

### cdb

Holds the SCSI control descriptor block bytes for this I/O.

### cdb_len

Number of bytes in the cdb; this can be a maximum of SCSI_MAX_CDB_LEN.
**io_id**
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**
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, `cdb_status` is set to `S_CHECK_CONDITION` to request an auto-sense request. The `scb->cdb_status` field 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` field.

**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 (number of bytes transferred = `bp->b_count - scb->data_resid`). Even if the I/O attempt is failed for some reason, set the `scb->data_resid` to indicate the number of bytes not yet transferred. Setting this field will have no adverse affect. The `scb->data_resid` field 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. The `scb->sense_status` field 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, `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`. The `scb->sense_bytes` field 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(9S), scsi_ifsw(9F)
NAME

scsi_action - Give I/O completion information to SCSI Services.

SYNOPSIS

#include <sys/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 specified 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 Minimum number of milliseconds before the I/O is retried.

DESCRIPTION

The 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 must 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.
The `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`.

>=0 Number of milliseconds before the I/O can be retried.

**CONSTRAINTS**

None

**SEE ALSO**

`biodone(9F), panic(9F)`
NAME

scsi_bus_lock - Acquire SCSI bus lock.

SYNOPSIS

#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. The scsi_bus_lock function can be called under interrupt context.

RETURN VALUES

None

CONSTRAINTS

Do not call 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(9F), scsi_dequeue_bp(9F), scsi_enqueue(9F)
NAME

scsi_bus_unlock - Release SCSI bus lock.

SYNOPSIS

#include <sys/scsi_ctl.h>

void scsi_bus_unlock(
    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.

RETURN VALUES

None

CONSTRAINTS

None

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(9F), scsi_dequeue_bp(9F), scsi_enqueue(9F)
NAME

scsi_cbfn - SCSI subsystem callback function.

SYNOPSIS

#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 after it calls
scsi_cbfn. Of course, the bp can 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/Os.

If the interface driver has attached a sense buffer to scb->sense_data, the sense_data buffer
must be valid until scsi_cbfn returns. The interface driver is forbidden from accessing the
sense data 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.

The scsi_cbfn function can be called either in process or interrupt context.

RETURN VALUES

None

CONSTRAINTS

The scsi_cbfn function must not be called with any locks held.

SEE ALSO

scb(9S), buf(9S)
NAME

scsi_cmd - Prepare driver-generated I/O requests.

SYNOPSIS

#include <sys/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, 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.

RETURN VALUES

N  Number of bytes transferred.
-1  Error.

CONSTRAINTS

The scsi_cmd function shall not be called holding a spinlock or in the interrupt context.

SEE ALSO

biowait(9F), scsi_ctl(7), scsi_init_inquiry_data(9F), scsi_cmdx(9F), scsi_strategy(9F)
scci_cmdx(9F)

NAME

scci_cmdx - Prepare driver-generated I/O requests.

SYNOPSIS

#include <sys/scci_ctl.h>

int scci_cmdx(
    dev_t dev,
    int flags,
    int cdb_len,
    ubit8 *cdb,
    int nbytes,
    void *addr,
    ubit32 msecs,
    ubit32 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, 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 scci_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 scci_strategy
routine, and cleans up when the I/O is completed.

The scci_cmdx routine is used by drivers to perform initialization or ioctl types of operations.
It is also used within SCSI Services to perform scci_init_inquiry_data, scci_mode_sense,
and scci_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.

The scci_cmdx function 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 scci_ctl(7).
The msecs parameter is assigned to sctl_io->max_msecs, which itself is assigned to scb->msecs. Similarly, 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.

**CONSTRAINTS**

Do not call the scsi_cmdx function while holding a spinlock or in the interrupt context.

**SEE ALSO**

biowait(9F), scsi_ctl(7), scsi_init_inquiry_data(9F), scsi_strategy(9F)
NAME

scsi_ddsw - SCSI device switch structure.

SYNOPSIS

#include <sys/scsi_ctl.h>

struct scsi_ddsw {
    u_char                      blk_major;
    u_char                      raw_major;
    int                         dd_lun_size;
    int                         (*dd_open)();
    void                      (*dd_close)();
    int                         (*dd_strategy)();
    int                         (*dd_read)();
    int                         (*dd_write)();
    int                         (*dd_ioctl)();
    struct buf                (*dd_start)();
    int                         (*dd_done)();
    int                         (*dd_pass_thru_okay)();
    int                         (*dd_pass_thru_done)();
    int                         (*dd_ioctl_okay)();
    struct status_action       *dd_status_list;
    int                         dd_status_cnt;
    ubit32                     *dd_flags;
    wsio_drv_info_t            *wsio_drv;
};

PARAMETERS

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 a driver-supplied routine.

dd_close() Pointer to a driver-supplied routine.

dd_strategy() Pointer to a driver-supplied routine.

dd_read() Pointer to a driver-supplied routine.

dd_write() Pointer to a driver-supplied routine.

dd_ioctl() Pointer to a driver-supplied routine.

dd_start() Pointer to a driver-supplied routine.

dd_done() Pointer to a driver-supplied routine.

dd_pass_thru_okay() Pointer to a driver-supplied routine.

dd_pass_thru_done() Pointer to a driver-supplied routine.

dd_ioctl_okay() Pointer to a driver-supplied routine.

dd_flags Flag bits. Currently only DD_DDG is 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 must have the standard driver routines restrict their operation to calling the appropriate SCSI Services routine. Special processing and customizing must be handled in the special dd_* routines.

EXAMPLE

The following examples show 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 */

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 */
    NULL,            /* dd_ioctl_okay */
    mydriver_dd_ioctl_okay, /* dd_ioctl_okay */
    NULL,            /* dd_status_list */
    sizeof(mydriver_dd_status_list),  /* dd_status_list */
    NULL,            /* dd_flag bits DD_DDG */
   .seconds,        /* dd_flag bits DD_DDG */
};
```

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 */
    NULL,            /* dd_ioctl_okay */
    mydriver_dd_ioctl_okay, /* dd_ioctl_okay */
    mydriver_dd_status_list,  /* dd_status_list */
    sizeof(mydriver_dd_status_list),  /* dd_status_list */
    NULL,            /* dd_flag bits DD_DDG */
};
```
&mydriver_wsio_info
}

CONSTRANTS
None

SEE ALSO

scsi_lun_open(9F)
NAME

scsi_dequeue - Remove I/O requests from queues maintained by SCSI Services.

SYNOPSIS

#include <sys/scsi_ctl.h>

struct buf *scsi_dequeue(
  struct **qp,
  int where
);

PARAMETERS

qp Pointer to the head of a list of I/O requests.
where Location to extract from. Valid values are HEAD and TAIL.

DESCRIPTION

The scsi_dequeue function extracts the I/O request at HEAD or TAIL of the list. The qp parameter is based on the value of where, and returns the bp.

RETURN VALUES

NULL Queue is empty.
<>0 Pointer to a buf (I/O request) structure.

CONSTRAINTS

This must be called with 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(9F), scsi_enqueue(9F)
**NAME**

scsi_dequeue_bp - Remove a specific I/O request from a specified queue maintained by SCSI services.

**SYNOPSIS**

```c
#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 `qp` queue. The function searches the I/O queue for the I/O request specified by the `bp`, and returns the I/O request if one is found.

**RETURN VALUES**

- `NULL` The `bp` structure was not found in queue.
- `<>NULL` Pointer to a `buf` (I/O request) structure.

**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(9F)`, `scsi_enqueue(9F)`
**NAME**

scsi_enqueue - Add buffer \( bp \) to a specified queue maintained by SCSI services.

**SYNOPSIS**

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

void scsi_enqueue(
    struct **qp,
    struct buf *bp,
    int where
);
```

**PARAMETERS**

- \( qp \) — A pointer to the head of a list of I/O requests.
- \( bp \) — Pointer to a \( buf \) structure to be added to the list.
- \( where \) — Location to insert the I/O request. Valid values are HEAD and TAIL.

**DESCRIPTION**

The `scsi_enqueue` function enqueues \( bp \) at the HEAD or TAIL of an I/O requests list; \( qp \) is a pointer to the head of a list of I/O requests. If \( where \) is HEAD, the \( bp \) is inserted at the head 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 end of select_q */
    scsi_bus_lock(busp);
    scsi_enqueue(&busp->select_q, TAIL);
    scsi_bus_unlock(busp);
    ....
}
```

**SEE ALSO**

`scsi_dequeue(9F), scsi_dequeue_bp(9F)`
NAME

csgi_ifsw - SCSI interface driver switch structure.

SYNOPSIS

#include <sys/scsi_ctl.h>

DESCRIPTION

The interface driver switch structure exports an interface driver entry points and operational parameters to the SCSI services layer. The interface driver attach routine must initialize the ifsw field of the isc_table_type entry to point to a scsi_isw structure.

STRUCTURE MEMBERS

The scsi_ifsw structure is defined in <sys/scsi_ctl.h>. The following table shows a list of driver-accessible fields and types in the scsi_ifsw structure:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_flags</td>
<td>ubit8</td>
</tr>
<tr>
<td>if_max_tag</td>
<td>ubit8</td>
</tr>
<tr>
<td>if_scb_size</td>
<td>unsigned int</td>
</tr>
<tr>
<td>if_lun_size</td>
<td>unsigned int</td>
</tr>
<tr>
<td>if_tgt_size</td>
<td>unsigned int</td>
</tr>
<tr>
<td>if_bus_size</td>
<td>unsigned int</td>
</tr>
<tr>
<td>if_open</td>
<td>int (*) ()</td>
</tr>
<tr>
<td>if_close</td>
<td>void (*) ()</td>
</tr>
<tr>
<td>if_start</td>
<td>void (*) ()</td>
</tr>
<tr>
<td>if_abort</td>
<td>int (*) ()</td>
</tr>
<tr>
<td>if_bdr</td>
<td>int (*) ()</td>
</tr>
<tr>
<td>if_reset_bus</td>
<td>int (*) ()</td>
</tr>
<tr>
<td>if_io_max_size</td>
<td>ubit32</td>
</tr>
<tr>
<td>if_beg_align</td>
<td>ubit32</td>
</tr>
<tr>
<td>if_end_align</td>
<td>ubit32</td>
</tr>
</tbody>
</table>

if_flags

Interface driver flags convey information to the SCSI services on what it supports and what not. The possible flags are:

- **IF_B2_LIST**: The interface driver supports handling of disksort merge buffers.
- **IF_BUS_TAGS**: A default flag.
- **IF_NO_TAGS**: The interface driver does not support tags.
- **IF_OWNS_TAGS**: The interface driver owns tagged queueing.
if_max_tag

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/Os to the bus at any given time (this includes untagged I/Os). Currently, the maximum value of a tag can be 254, and the maximum value of if_max_tag is 255.

if_scb_size

The number of bytes the SCSI subsystem must 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’d by services for each I/O attempt prior to putting the I/O on the select queue. The if_scb area is not touched by services at any other time.

if_lun_size

The number of bytes the SCSI subsystem must allocate and attach to each scsi_lun structure for use by the interface driver. The if_lun field of the 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 must allocate and attach to each scsi_tgt structure for use by the interface driver. The if_tgt field of the scsi_tgt structure is a pointer to ifsw->if_tgt_size bytes for the use of the interface driver.

if_bus_size

The number of bytes the SCSI subsystem must allocate and attach to each scsi_bus structure for use by the interface driver. The if_bus field of the 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 errored back to 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(9S), isc_table_type(9S), scb(9S)
NAME

scsi_init_inquiry_data - Perform the first inquiry request on a device.

SYNOPSIS

#include <sys/scsi_ctl.h>

int scsi_init_inquiry_data(
    dev_t dev
);

PARAMETERS

dev Device number

DESCRIPTION

In the following description, lp refers to a pointer to the scsi_lun structure.

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.

CONSTRAINTS

None
NAME

scsi_ioctl - Standard SCSI ioctl routine.

SYNOPSIS

#include <sys/scsi_ctl.h>

int scsi_ioctl(
    dev_t dev,
    int cmd,
    caddr_t data,
    int flags
);

PARAMETERS

dev  Device number

cmd  The ioctl command. It can be one of the commands listed in the DESCRIPTION section 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. The ioctl commands that 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.

The 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 blocks into a structure of type capacity_type. Returns information from data saved in the LUN structure during the open. The capacity_type structure type is defined in <sys/diskio.h>.

DIOC DESCRIBE

ioctl(fd, DIOC DESCRIBE, &describe)

Returns information about the device into a structure of type describe_type. 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 returned information from data is saved in the LUN structure during the open. The describe_type structure type is defined in <sys/diskio.h>.

DIOC GET_PFTIMEOUT

ioctl(fd, DIOC GET_PFTIMEOUT, &msec)

Returns an integer value (msec) used for timing all LVM requests within bp->b_flags and B_PFTIMEOUT. The msec parameter is of type INT.
DIOC_RSTCLR

    ioctl(fd, DIOC_RSTCLR)

Perform Bus Device Reset for SCSI device.

DIOC_SET_PFTIMEOUT

    ioctl(fd, DIOC_SET_PFTIMEOUT, &msecs)

Sets an integer value (msecs) used for timing all LVM requests within bp->bflags & B_PFTIMEOUT; zero (0) means reset to driver's default. The msecs parameter is of the type INT.

SIOC_CAPACITY

    ioctl(fd, SIOC_CAPACITY, &capacity)

Returns device media capacity and device block size information from data saved in the LUN structure during the open. The capacity_type structure type is defined in <sys/scsi.h>.

SIOC_CMD_MODE

    ioctl(fd, , &mode)

Used by either the device driver or the pass-through driver, sctl.

The mode parameter can be either 0 (off) or 1 (on).

Error Codes:

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    The mode parameter is invalid.

SIOC_FORMAT

    ioctl(fd, SIOC_FORMAT, &format)

Format device media capacity and block size. The sioc_format structure type is defined in <sys/scsi.h>.

Error Codes:

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.

SIOC_GET_BUS_LIMITS

    ioctl(fd, limits, &limits)

If limits have not been set, the act of getting them, sets them.

The sioc_bus_limits structure type is defined in <sys/scsi.h>. See scsi_ctl(7).

Error Codes:

None

SIOC_GET_BUS_PARMS

    ioctl(fd, parms, &parms)

The sioc_bus_parms structure type is defined in <sys/scsi.h>. See scsi_ctl(7).

Error Codes:
None

**SIOC_GET_LUN_LIMITS**

```c
ioctl(fd, SIOC_GET_LUN_LIMITS, &lun_limits)
```

If limits have not been set, the act of getting them sets them.

The `sioc_lun_limits` structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7)`.

**Error Codes:**

None

**SIOC_GET_LUN_PARMS**

```c
ioctl(fd, SIOC_GET_LUN_PARMS, &lun_parms)
```

The `sioc_lun_parms` structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7)`.

**Error Codes:**

None

**SIOC_GET_TGT_LIMITS**

```c
ioctl(fd, limits, &limits)
```

If limits have not been set, the act of getting them, sets them.

The `sioc_tgt_limits` structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7)`.

**Error Codes:**

None

**SIOC_GET_TGT_PARMS**

```c
ioctl(fd, SIOC_GET_TGT_PARMS, &tgtParms)
```

The `sioc_tgt_parms` structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7)`.

**Error Codes:**

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 `inquiry` and `inquiry_2` structures types, and `inquiry_data` union are defined in `<sys/scsi.h>`.

**SIOC_EXCLUSIVE**

```c
ioctl(fd, SIOC_EXCLUSIVE, &mode)
```

Gain/release exclusive access mode.

The `mode` parameter is an integer that can contain one of the following values:

<table>
<thead>
<tr>
<th><code>mode</code></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 Codes:**

- EBUSY Other opens are active on the level for which exclusive access is desired (LUN, target, and bus).
EINVAL  The *mode* parameter is not in the range 0 to 5, inclusive.

**SIOC_IO**

```c
ioctl(fd, SIOC_IO, &sctl)
```

This is used for pass-through I/Os.

The *sctl* 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: 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.

**NOTE:** Any data structure referenced by *arg* must not contain any pointers.

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 not to initiate SCSI wide data transfer negotiation. *SCTL_INIT_SDTR* specifies to initiate SCSI synchronous data transfer negotiation. 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 must 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 must initiate SDTR negotiation immediately following the WDTR negotiation.

If *SCTL_INIT_SDTR* is set and *SCTL_INIT_WDTR* is not, the interface driver must 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 must 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.

**Error Codes:**

EACCES  The user is not superuser or there is no write access permission.

**SIOC_PRIORITY_MODE**

```c
ioctl(fd, SIOC_PRIORITY_MODE, &mode)
```

See *scsi_ctl*(7).
A device can only be put into priority mode from the pass-through driver. 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 that 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 occurs. Therefore, the process 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 mode parameter is an integer value. 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   The int parameter is invalid, or the command was not invoked by the pass-through major number.

**SIOC_RESET_BUS**

```c
ioctl(fd, SIOC_RESET_BUS)
```

This command causes the SCSI RST line to be pulled by calling the interface driver's if_reset_bus routine.

**Error Codes:**

EACCES   The user is not superuser.
EINVAL   The if_reset_bus routine is NULL.

**SIOC_RESET_DEV**

```c
ioctl(fd, SIOC_RESET_DEV)
```

This command causes a SCSI Bus Device Reset to be sent to the target device by calling the interface driver if_bdr routine.

**Error Codes:**

EACCES   The user is not superuser.
EINVAL   The if_bdr routine is NULL.

**SIOC_RETURN_STATUS**

```c
ioctl(fd, SIOC_RETURN_STATUS, &status)
```

Can be used by either device driver or sctl pass-through driver, whether in command mode or not. The SCSI status is 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 (SCSI status; does not include SCTL_xxx).

For the 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. The x[CMD_MODE_ONLY] field 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.

**Error Codes:**

None

`SIOC_SET_BUS_LIMITS`

`ioctl(fd, SIOC_SET_BUS_LIMITS, &limits)`

The sioc_bus_limits structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7).

**Error Codes:**

EACCES If not superuser or write permission.
EINV AL If reserved fields are not zero.

`SIOC_SET_CMD`

`ioctl(fd, SIOC_SET_CMD, &parms)`

This command can 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 scsi_cmd_parms structure type is defined in `<sys/scsi.h>`.

**Error Codes:**

EACCES The command was not invoked by the “command code major”.
EINV AL Theparms->cmd_type parameter is less than 1 or greater than SCSI_MAX_CDB_LEN.

`SIOC_SET_LUN_LIMITS`

`ioctl(fd, LIMITS, &limits)`

The sioc_lun_limits structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7).

**Error Codes:**

EACCES If not superuser or write permission.
EINV AL If reserved fields are not zero.

`SIOC_SET_TGT_LIMITS`

`ioctl(fd, SIOC_SET_TGT_LIMITS, &limits)`

The sioc_tgt_limits structure type is defined in `<sys/scsi.h>`. See `scsi_ctl(7).

**Error Returns:**

EACCES If not superuser or write permission.
EINV AL If reserved fields are not zero.

`SIOC_XSENSE`

`ioctl(fd, SIOC_XSENSE, &data)`

The last available sense data is copied to the passed sense_data structure. It can 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. The x[CMD_MODE_ONLY] field 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, xsense, xsense_aligned, and sense_data data structures are defined in the `<sys/scsi.h>` header file.
Error Codes:
  EINVAL  The data size is not equal to 128 bytes.

RETURN VALUES
0  Successful completion.
-1  Error.

CONSTRAINTS
Do not call in interrupt context or while holding spinlocks.
NAME

scsi_lun_close - Close a device.

SYNOPSIS

#include <sys/scsi_ctl.h>

void scsi_lun_close(
  dev_t dev
);

PARAMETERS

dev 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 and close semaphore.

The 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.

RETURN VALUES

None

CONSTRAINTS

Call this routine only when dev is open. If it is not, the system panics in m_scsi_lun when it attempts to dereference a NULL pointer.

Do not call in interrupt context or while holding spinlocks.
NAME
scsi_lun_open - Open the elements of the hardware path of a SCSI LUN.

SYNOPSIS
#include <sys/scsi_ctl.h>

int scsi_lun_open(
    dev_t dev,
    struct scsi_ddsw *ddsw,
    int oflags
);

PARAMETERS
dev   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 and close semaphore.

Uses kmalloc to allocate memory for the scsi_lun structure.

Checks if major(dev) == sctl_ddsw.raw_major.

RETURN VALUES
0       Successful completion.
[EBUSY] The LUN is already opened EXCLUSIVE by another or the open is incomplete and this is not a pass-through or command mode open attempt.
[EINVAL] The open request major number does not make sense.
[ENXIO] The LUN requested is greater than SCSI_MAX_LUN_ID.

Other errors may be returned from ddswn->dd_open, if_open, scsi_bus_open, or scsi_tgt_open, if they are called.

CONSTRAINTS
Do not call in interrupt context or while holding spinlocks.
NAME

scsi_read - Read from device.

SYNOPSIS

#include <sys/scsi_ctl.h>

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. Return an errno value.

CONSTRAINTS

Do not call in interrupt context or while holding spinlocks.

SEE ALSO

physio(9F), scsi_ddsw(9S)
scsi_sense_action(9F)

NAME

scsi_sense_action - Decode SCSI sense information.

SYNOPSIS

#include <sys/scsi_ctl.h>

int scsi_sense_action(
    struct buf *bp,
    struct sense_action *sense_list,
    size_t n
);

PARAMETERS

bp Pointer to the I/O buf structure.
sense_list List of actions to take.
n Number of actions in the list.

DESCRIPTION

The scsi_sense_action SCSI function decodes SCSI sense information. It traverses the functions in a driver's sense action list trying to find a match, and calls the associated action function. It provides the very valuable service of interpreting sense data with regard to SCSI, CCS, or SCSI-2 compliance, so the device driver does not need to worry about such things.

The scsi_sense_action function must 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 have any protection concerns.

The inquiry data for the device must be initialized with scsi_init_inquiry_data before it can be interpreted. If an I/O completes before the inquiry data is initialized, scsi_sense_action will not match anything other than wild card entries.

RETURN VALUES

0 Successful completion.
<>0 Error. The value is provided by the sense action called.

CONSTRAINTS

None

SEE ALSO

scsi_init_inquiry_data(9F)
NAME

scsi_strategy - Enqueue the bp to await resources.

SYNOPSIS

#include <sys/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.

The scsi_strategy function 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.

The scsi_strategy function must be invoked with a valid bp. If a special request (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: The scsi_strategy function calls dd_strategy, if present, holding the lun_lock.

RETURN VALUES

None

CONSTRAINTS

This routine cannot block.

Do not call while holding spinlocks.

SEE ALSO

biodone(9F), scsi_enqueue(9F)
NAME

scsi_write - Write to device.

SYNOPSIS

#include <sys/scsi_ctl.h>

int scsi_write(
    dev_t dev,
    struct uio *uiop
);

PARAMETERS

dev    Device number.
uiop   A structure containing transfer information.

DESCRIPTION

The scsi_write SCSI function is used for normal (synchronous) writes and for command mode
I/Os for which the SCSI_CMD_MODE ioctl, 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. Return an errno value.

CONSTRAINTS

Do not call in interrupt context or while holding spinlocks.

SEE ALSO

physio(9F), scsi_ddsw(9S)
This appendix contains reference pages that have been deprecated in HP-UX 11i. Developers are encouraged to use the newer interfaces. For information about these interfaces, see the *HP-UX 11i v1 Driver Development Reference* at: http://www.hp.com/go/hpux_ddk

Table A-1 Deprecated Interfaces, Their Description, and Replacement

<table>
<thead>
<tr>
<th>Deprecated Interface</th>
<th>Description</th>
<th>Replacement Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_map_context</td>
<td>Macro to initialize mapping context structure.</td>
<td>N/A</td>
</tr>
<tr>
<td>isrlink</td>
<td>Register an interrupt service routine.</td>
<td>wsio_intr_alloc</td>
</tr>
<tr>
<td>isrunlink</td>
<td>Remove the ISR registered by isrlink.</td>
<td>wsio_intr_deactivate wsio_intr_free</td>
</tr>
<tr>
<td>m_instance</td>
<td>Get the device instance field from the device number.</td>
<td>wsio_isc_to_instance</td>
</tr>
<tr>
<td>pci_get_fru_info_isc</td>
<td>Get Field Replaceable Unit (FRU) information for the device associated with an ISC.</td>
<td>N/A</td>
</tr>
<tr>
<td>wsio_allocate_shared_memory</td>
<td>Allocate and map contiguous memory used for continuous DMA.</td>
<td>wsio_allocate_shared_mem</td>
</tr>
<tr>
<td>wsio_fastmap</td>
<td>Map all or part of a host address range into an I/O virtual address range.</td>
<td>wsio_fastmap_dma_buffer</td>
</tr>
<tr>
<td>wsio_flush_shared_memory</td>
<td>Flush the memory previously allocated and mapped by wsio_allocate_shared_memory.</td>
<td>wsio_flush_shared_memory</td>
</tr>
<tr>
<td>wsio_free_shared_memory</td>
<td>Release and unmap contiguous memory previously allocated and mapped by wsio_allocate_shared_memory.</td>
<td>wsio_free_shared_memory</td>
</tr>
<tr>
<td>wsio_get_attribute</td>
<td>Gets an attribute registered with an interface.</td>
<td>prop_get</td>
</tr>
<tr>
<td>wsio_get_interrupts</td>
<td>Determine which interrupt has been assigned to a card.</td>
<td>wsio_get_irq_line</td>
</tr>
<tr>
<td>wsio_get_pva</td>
<td>Translate an I/O virtual address to its processor virtual address.</td>
<td>wsio_iova_to_phys</td>
</tr>
<tr>
<td>wsio_get_registers</td>
<td>Get the register addresses of an interface card.</td>
<td>wsio_get_all_registers</td>
</tr>
<tr>
<td>wsio_install_drv_func</td>
<td>Register a driver function with the WSIO driver environment.</td>
<td>wsio_install_drv_event_handler</td>
</tr>
<tr>
<td>wsio_map</td>
<td>Map all or part of a host address range into an I/O virtual address range.</td>
<td>wsio_map_dma_buffer</td>
</tr>
<tr>
<td>wsio_register_probe_func</td>
<td>Insert a driver-specified probe function into the global probe list.</td>
<td>wsio_register_dev_probe wsio_register_addr_probe</td>
</tr>
<tr>
<td>wsio_remap</td>
<td>Map a host range into a pre-mapped I/O Virtual Address range.</td>
<td>wsio_remap_dma_buffer</td>
</tr>
<tr>
<td>wsio_set_attributes</td>
<td>Set map function attributes.</td>
<td>wsio_dma_set_device_attributes wsio_set_dma_attributes</td>
</tr>
</tbody>
</table>