5 PCI Reference Pages

PCI Macros and Functions

This chapter describes driver functions that are specific to PCI Services.

In the function synopses, each parameter type is prefixed with the comment /*IN*/ or /*OUT*/, indicating that the parameter is input to or output from the function. These comments are included here because they are helpful and because they are used as keywords on other platforms.

PCI Masters and Coherency

Be aware that certain combinations of WSIO mapping service calls can interact with PCI masters to create an inconsistent view of memory. This behavior is documented in pci_errata (PCI5).
NAME

CONNECT_INIT_ROUTINE (PCI3) – Associate an initialization routine with a driver.

SYNOPSIS

#include <sys/pci.h>

#define CONNECT_INIT_ROUTINE(isc, initRoutine) \
((isc)->gfsw->init = (initRoutine))

PARAMETERS

isc Pointer to an ISC table associated with the device.
init Pointer to your driver's initialization routine.

DESCRIPTION

CONNECT_INIT_ROUTINE() is a macro that associates a driver_if_init() routine with a driver. It is typically used in a device's driver_attach() routine. After all attach routines are executed, the driver_if_init() routine is called by the kernel to perform further card and driver initialization.

If the driver performs all needed initialization in its driver_attach() routine, a driver_if_init() routine is not needed.
NAME

PCI_ATTACH_DEV_INIT_ERROR (PCI3) – Report initialization error to WSIO Services.

SYNOPSIS

#include <sys/pci.h>

#define PCI_ATTACH_DEV_INIT_ERROR(isc)  
  (((struct wsio_if_info *)((isc)->if_info))->flags |= INIT_ERROR)

PARAMETERS

isc  Pointer to the ISC structure passed into your attach routine.

DESCRIPTION

PCI_ATTACH_DEV_INIT_ERROR() is a macro that reports to WSIO Services that an error occurred during the device’s initialization. It causes an error flag to be set in a structure in isc. It should be used as appropriate in the device’s driver_attach() or driver_if_init() routine.
NAME

pci_desc_bus_transactions_isc (PCI3) – Describe the typical bus performance path transaction size.

SYNOPSIS

#include <sys/pci.h>

int pci_desc_bus_transactions_isc (    /*IN*/ struct isc_table_type * isc,    /*IN*/ BUS_TRANS_DESC * desc);

PARAMETERS

isc Pointer to an ISC table associated with the device.
desc A pointer to a BUS_TRANS_DESC structure, defined as:

typedef struct bus_trans_desc
    {
    uint32_t read_width;
    uint32_t write_width;
    uint8_t reserved[20];
    } BUS_TRANS_DESC;

where:

read_width The number of 32-bit words in the width of the read path. 0 means don’t change the current value.

write_width The number of 32-bit words in the width of the write path. 0 means don’t change the current value.

reserved Reserved for future extensions.

DESCRIPTION

The pci_desc_bus_transactions_isc() PCI function allows a driver to describe the bus transaction size of a card's typical performance DMA accesses. Its use is entirely optional, since PCI Services provides a reasonable, general-purpose default.

The key to understanding how to use this routine for performance tuning is to recognize that the PCI bus supports variable-length data transactions. These transaction lengths may not map directly to transaction lengths on other busses on the system. Also, performance depends on other 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 don’t use pci_desc_bus_transactions_isc(), PCI Services provide defaults that are intended to be safe and to give reasonable performance.
RETURN VALUES

PCI_OKAY
The hints were used.

PCI_BUS_HINTS_BAD_DATA
The BUS_TRANS_DESC structure is incorrect.

PCI_BUS_HINTS_NOT_USED
The hints are not supported in this configuration.

CONSTRAINTS
NAME

`pci_get_fru_info_isc` (PCI3) – Get field-replaceable-unit (FRU) information for the device associated with an ISC.

SYNOPSIS

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

int pci_get_fru_info_isc (
    /*IN*/ struct isc_table_type *isc,
    /*OUT*/ int *fru_info);
```

PARAMETERS

- **isc** Pointer to an ISC table structure associated with the device.
- **fru_info** A pointer to the location where the routine should place the FRU information.

DESCRIPTION

The `pci_get_fru_info_isc()` PCI function returns field-replaceable-unit (FRU) information for the device associated with an ISC.

The FRU information is the physical location of the device on a particular machine. To be able to provide FRU information for a device, the hardware of the machine it is on must be able to “see” it (meaning it must be either a built-in device or in a slot directly attached to the machine).

If the device is a card in an expansion slot, the FRU number is the slot number on the machine. If the device is built-in, the FRU number is the built-in device number, provided one was assigned to it by the manufacturer. Otherwise, the device must be located on an expansion bus.

RETURN VALUES

- **PCI_GET_FRU_INFO_BUILT_IN_FRU**
  The device is built-in and was assigned the built-in device number given in `fru_info`.

- **PCI_GET_FRU_INFO_BUILT_IN_NO_FRU**
  The device is built-in but was not assigned a built-in device number. `fru_info` is not valid.

- **PCI_GET_FRU_INFO_EXPANSIONDEVICE**
  The device is an expansion device located in the slot number given in `fru_info`.

- **PCI_GET_FRU_INFO_DEV_NOT_FOUND**
  The device is neither built-in nor found in any expansion slot; it is not in any physical location that the hardware knows about. It must, therefore, be located on a bus that is downstream of a PCI-to-PCI bridge. `fru_info` is not valid.

- **PCI_GET_FRU_INFO_NOT_IMPLEMENTED**
  This functionality is not available for the device. `fru_info` is not valid.
CONSTRAINTS
NAME

pci_get_port_hndl_isc (PCI3) – Obtain a system-defined handle for manipulating a range of PCI I/O ports.

SYNOPSIS

#include <sys/pci.h>

int
pci_get_port_hndl_isc (/*IN*/ struct isc_table_type * isc,
/*IN*/ uint32_t   pci_io_addr,
/*IN*/ uint32_t   size,
/*OUT*/ PCI_PORT_HNDL * phndl);

PARAMETERS

isc         Pointer to an ISC table associated with the device.
pci_io_addr The address of a range of PCI I/O ports.
size        The size of the PCI I/O ports.
phndl       A pointer to the location where the routine should place the handle provided by the system.

DESCRIPTION

The pci_get_port_hndl_isc() PCI function obtains a system-defined handle for manipulating a range of PCI I/O ports.

The routine can block or sleep and, therefore, should only be called in a thread context.

RETURN VALUES

0         Failure. A handle could not be returned by the system.
1         Success. The value pointed to by phndl is a valid handle.

CONSTRAINTS
EXAMPLES

The `pci_read_cfg_uint32_isc()` routine reads the card's configuration space and retrieves the PCI address associated with an I/O port range. This address and size is passed into `pci_get_port_hndl_isc()` to get a handle. The handle is needed to access the port through the `pci_read_port_uint32_isc()` function.

```c
PCI_PORT_HNDL phndl;
uint32_t pci_port_addr;
uint32_t data;

/*
 * get the io port address and mask off unwanted bottom
 * bits
 */
pci_read_cfg_uint32_isc(isc, mydriver_PORT_BASE_REG,
 &pci_port_addr);
pci_port_addr &= ~0x3;

/*
 * get the port handle
 */
if (pci_get_port_hndl_isc(isc, pci_port_addr, mydriver_PORT_BLOCK_SIZE, &phndl)) {
    /*
     * use it for as long as you want,
     * then return it when it is no longer needed
     */
    pci_read_port_uint32_isc(isc, phndl, mydriver_PORT_OFFSET,
     &data);
    ...                                                                                                                                                                                                                                                                                                                                                           
    pci_unget_port_hndl_isc(isc, pci_port_addr, mydriver_PORT_BLOCK_SIZE, phndl);
}
```

SEE ALSO

`pci_unget_port_hndl_isc` (PC13)
NAME

pci_read_cfg_uintN_isc (PCI3) – Read unsigned integer from a PCI configuration register.

SYNOPSIS

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

void
pci_read_cfg_uint8_isc (/*IN*/ struct isc_table_type * isc,
                         /*IN*/ int reg_num,
                         /*OUT*/ uint8_t * data_read);

void
pci_read_cfg_uint16_isc (/*IN*/ struct isc_table_type * isc,
                          /*IN*/ int reg_num,
                          /*OUT*/ uint16_t * data_read);

void
pci_read_cfg_uint32_isc (/*IN*/ struct isc_table_type * isc,
                          /*IN*/ int reg_num,
                          /*OUT*/ uint32_t * data_read);
```

PARAMETERS

- **isc**
  Pointer to an ISC table associated with the device.

- **reg_num**
  The offset of a PCI configuration register for the PCI device/function specified by *isc*. It can be a PCI_CS_* constant, defined in pci.h.

- **data_read**
  A pointer to an 8-, 16-, or 32-bit location where the routine should place the value.

DESCRIPTION

The **pci_read_cfg_uintN_isc** PCI functions read an 8-, 16-, or 32-bit unsigned integer from a PCI configuration register for a particular PCI device/function.

RETURN VALUES

The **pci_read_cfg_uintN_isc** routines do not return values.

CONSTRAINTS
EXAMPLE

#include <sys/pci.h>

static void
mydriver_set_io_master (struct isc_table_type * isc)
{
    unsigned short hwid;
    uint16_t old_cmdreg;
    PCI_PORT_HNDL ph;

    pci_read_cfg_uint16_isc(isc, PCI_CS_COMMAND, &old_cmdreg);
    pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND, old_cmdreg |
               PCI_CMD_IO_SPACE | PCI_CMD_BUS_MASTER);

    ...
}

SEE ALSO

pci_write_cfg_uintN_isc (PCI3)
NAME

pci_read_port_uintN_isc (PCI3) – Read little-endian data from an I/O port.

SYNOPSIS

#include <sys/pci.h>

void
pci_read_port_uint8_isc (
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*OUT*/ uint8_t *data);

guid
pci_read_port_uint16_isc (
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*OUT*/ uint16_t *data);

guid
pci_read_port_uint32_isc (
    /*IN*/ struct isc_table_type *isc,
    /*IN*/ PCI_PORT_HNDL ph,
    /*IN*/ uint32_t offset,
    /*OUT*/ uint32_t *data);

PARAMETERS

isc Pointer to an ISC table associated with the device.

ph A port handle previously obtained with a call to pci_get_port_hndl_isc().

offset An offset from ph.

data A pointer to an 8-, 16-, 32-bit location where the routine should place the value.

DESCRIPTION

The pci_read_port_uintN_isc PCI functions read 8-, 16-, or 32-bit little-endian data for the device/function specified by isc from the I/O port represented by the PCI port handle ph and offset offset. You will probably need to swap bytes if your driver will operate on 16, or 32 bit data.

RETURN VALUES

The pci_read_port_uintN_isc() routines do not return values.

CONSTRAINTS
EXAMPLES

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

#define MY_IOMAP_BASE 0x10
#define MY_PORT_SIZE 0x100
#define MY_IDREG 0x0
#define MY_HVID 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_HVID) != MY_HVID) {
      return -1;
    }
    isc->mapped=(int)ph.hndl;
  } else {
    msg_printf("pci_get_port_hndl_isc() failed\n");
    return -1;
  }
  return 0;
}
```

SEE ALSO

`pci_write_port_uintN_isc` (PCI3)
NAME

pci_unget_port_hndl_isc (PCI3) – 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 (  
    /*IN*/ struct isc_table_type *isc,  
    /*IN*/ uint32_t          pci_io_addr,  
    /*IN*/ uint32_t          size,  
    /*IN*/ PCI_PORT_HNDL     phndl);  

PARAMETERS

isc Pointer to an ISC table associated with the device.
pci_io_addr The address of a range of PCI I/O ports.
size The size of the PCI I/O ports.
phndl A handle obtained for these parameters by a previous call to pci_get_port_hndl_isc().

DESCRIPTION

The pci_unget_port_hndl_isc() PCI function deletes a system-defined handle for manipulating a range of PCI I/O ports.

RETURN VALUES

0 Failure. The handle could not be deleted by the system.
1 Success. The handle was deleted by the system.

CONSTRAINTS

EXAMPLES

See pci_get_port_hndl_isc (PCI3).

SEE ALSO

pci_get_port_hndl_isc (PCI3)
NAME

pci_write_cfg_uintN_isc (PCI3) – Write unsigned integer to a PCI configuration register.

SYNOPSIS

#include <sys/pci.h>

void
pci_write_cfg_uint8_isc (  
  /*IN*/ struct isc_table_type *isc, 
  /*IN*/ int reg_num, 
  /*IN*/ uint8_t data_write); 

void
pci_write_cfg_uint16_isc (  
  /*IN*/ struct isc_table_type *isc, 
  /*IN*/ int reg_num, 
  /*IN*/ uint16_t data_write); 

void
pci_write_cfg_uint32_isc (  
  /*IN*/ struct isc_table_type *isc, 
  /*IN*/ int reg_num, 
  /*IN*/ uint32_t data_write); 

PARAMETERS

isc Pointer to an ISC table associated with the device.
reg_num The number of a PCI configuration register for the PCI device/function specified by isc. It can be a PCI_CS_* constant, defined in pci.h.
data_write The 8-, 16-, or 32-bit value to be written.

DESCRIPTION

The pci_write_cfg_uintN_isc() PCI functions write an 8-, 16-, or 32-bit unsigned integer to a PCI configuration register for a particular PCI device or function.

RETURN VALUES

The pci_write_cfg_uintN_isc() routines do not return values.

CONSTRAINTS

SEE ALSO

pci_read_cfg_uintN_isc (PCI3)
NAME

pci_write_port_uintN_isc (PCI3) – Write little-endian data to an I/O port.

SYNOPSIS

#include <sys/pci.h>

void
pci_write_port_uint8_isc (/*IN*/ struct isc_table_type *isc,
                         /*IN*/ PCI_PORT_HNDL ph,
                         /*IN*/ uint32_t offset,
                         /*IN*/ uint8_t data);

void
pci_write_port_uint16_isc (/*IN*/ struct isc_table_type *isc,
                           /*IN*/ PCI_PORT_HNDL ph,
                           /*IN*/ uint32_t offset,
                           /*IN*/ uint16_t data);

void
pci_write_port_uint32_isc (/*IN*/ struct isc_table_type *isc,
                           /*IN*/ PCI_PORT_HNDL ph,
                           /*IN*/ uint32_t offset,
                           /*IN*/ uint32_t data);

PARAMETERS

isc Pointer to an ISC table associated with the device.

ph A port handle previously obtained with a call to pci_get_port_hndl_isc().

offset An offset from ph.

data The 8-, 16-, or 32-bit value to be written.

DESCRIPTION

The pci_write_port_uintN_isc() PCI functions write 8-, 16-, or 32-bit little-endian data for the device/function specified by isc to the I/O port represented by the PCI port handle ph and offset offset. You will probably need to swap bytes if your driver is operating on 16- or 32-bit data.

RETURN VALUES

The pci_write_port_uintN_isc() routines do not return values.

CONSTRAINTS
SEE ALSO

pci_read_port_uintN_isc (PCI3)
NAME

READ_REG_UINTn_ISC (PCI3) – Read and byte-swap data from a little-endian bus.

SYNOPSIS

#include <sys/pci.h>
void
READ_REG_UINT8_ISC ( /*IN*/ struct isc_table_type *isc,
                     /*IN*/ uint8_t *addr,
                     /*OUT*/ uint8_t *data);

void
READ_REG_UINT16_ISC ( /*IN*/ struct isc_table_type *isc,
                       /*IN*/ uint16_t *addr,
                       /*OUT*/ uint16_t *data);

void
READ_REG_UINT32_ISC ( /*IN*/ struct isc_table_type *isc,
                       /*IN*/ uint32_t *addr,
                       /*OUT*/ uint32_t *data);

PARAMETERS

isc  Pointer to an ISC table associated with the driver.

addr A pointer to the address of the bus data. It must be one of:

- A virtual address mapped with map_mem_to_host().
- A mapped offset in the automatically mapped first-base-address register range contained in isc->if_reg_ptr.
  - Only the first nonzero 32-bit-wide memory base register found can be mapped, starting in the range 0x10 and searching up through 0x24 (the six possible base address register locations in configuration space).
  - If that base-register’s size is in excess of 8 KB, it is not mapped and isc->if_reg_ptr is set to NULL. In this case, the driver must map the base register it wants to use.
- The address of a memory buffer shared between the driver and a little-endian bus master.

data A pointer to an 8-, 16-, or 32-bit location where the routine should place the resultant data.
DESCRIPTION

The `READ_REG_UINTn_ISC()` PCI services are macros that read and byte-swap data located at `addr` from a little-endian bus and place it in `data`.

If the PCI adapter that your card is running under has directly mapped the PCI memory space into driver-accessible system I/O space, you can improve the performance of `READ_REG_UINTn_ISC()` if you define the flag `PCI_LITTLE_ENDIAN_ONLY` prior to including the `pci.h` header file. This causes `READ_REG_UINTn_ISC()` to perform a simple byte swap instead of calling a function that tests byte ordering.

RETURN VALUES

The `READ_REG_UINTn_ISC()` routines do not return values.

CONSTRAINTS

EXAMPLES

```c
#include <sys/pci.h>
define MY_REGISTER_OFFSET 0x40
/* the address of some register on my card */

uint8_t data8;
uint8_t *addr = isc->if_reg_ptr + MY_REGISTER_OFFSET;
    /* virtual address plus an offset */

/*
 * code accessing registers is expanded inline
 */
READ_REG_UINT8_ISC(isc, addr, &data8);
```

SEE ALSO

`WRITE_REG_UINTn_ISC` (PCI3)
NAME

WRITE_REG_UINTn_ISC (PCI3) – Byte-swap and write data to a little-endian bus.

SYNOPSIS

#include <sys/pci.h>

void
WRITE_REG_UINT8_ISC(  
  /*IN*/ struct isc_table_type *isc,  
  /*IN*/ uint8_t *addr,  
  /*IN*/ uint8_t data);

void
WRITE_REG_UINT16_ISC(  
  /*IN*/ struct isc_table_type *isc,  
  /*IN*/ uint16_t *addr,  
  /*IN*/ uint16_t data);

void
WRITE_REG_UINT32_ISC(  
  /*IN*/ struct isc_table_type *isc,  
  /*IN*/ uint32_t *addr,  
  /*IN*/ uint32_t data);

PARAMETERS

isc Pointer to an ISC table associated with the device.

data 8-, 16-, or 32-bit data to be written.

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.
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 flag PCI_LITTLE_ENDIAN_ONLY prior to including the pci.h header file. This causes WRITE_REG_UINTn_ISC() to perform a simple byte swap instead of calling a function that tests byte ordering.

RETURN VALUES

The WRITE_REG_UINTn_ISC() routines do not return values.

CONSTRAINTS

EXAMPLES

#include <sys/pci.h>
define MY_REGISTER_OFFSET 0x40

uint8_t data8;
uint8_t *addr = isc->if_reg_ptr + MY_REGISTER_OFFSET;
    /* virtual address plus an offset */

/*
 * code accessing registers is expanded inline
 */
WRITE_REG_UINT8_ISC(isc, addr, &data8);

SEE ALSO

READ_REG_UINTn_ISC (PCI3)
NAME

PCI_ERRATA-1 (PCI5) –

MEMORY COHERENCY ISSUES

Certain combinations of WSIO mapping service calls can interact with PCI masters on C class and J class
processors to create an inconsistent view of memory.

It is possible for prefetching of host memory by the PA hardware chipsets to result in a PCI master reading
stale data, even though the proper dma_sync() calls have been made. The problem does NOT occur if:

1. The PCI master does normal Memory Read transactions. i.e., the master does not master Memory Read
   Multiple (MRM) or Memory Read Line (MRL) transactions.
2. The mapping is done with wsio_map() with flags IO_NO_SEQ and IO_SAFE set, regardless of the type of
   transactions the PCI master uses.
3. The mapping is done with wsio_fastmap() and the PCI master does NOT use MRM or MRL
   transactions.

DETAILS

There are two hardware prefetch buffers in the PA hardware chipset between memory and any PCI device.
One is in the system’s PCI bridge chip, and the other is system’s GSC I/O bridge chip (which connects to the
PA side of the PCI bridge chip). The I/O bridge chip has a cache line size prefetch buffer for each I/O (GSC)
slot.

For the following discussion assume that an I/O TLB was mapped using either wsio_fastmap() or wsio_map
without IO_NO_SEQ and IO_SAFE flag bits set.

When a PCI bus master runs an MRM or MRL transaction, the following events happen:

1. The PCI bridge chip requests a cache line (8 words), starting at the PCI master requested start address,
   from the I/O bridge chip. Since the PCI master is running an MRM or MRL transaction, this request is
   made with a prefetch hint enabled for the I/O bridge chip. In addition it requests subsequent cache lines
   from the I/O bridge chip, with the exact number of extra lines dependent upon whether an MRM or MRL
   PCI transaction is in progress, and whether or not the end of a physical page is near (the PCI bridge chip
   will not prefetch past the end of a page).
2. The I/O bridge chip, for each cache line requested, fills the request immediately from its own prefetch
   buffer if the requested line resides there, or gets the cache line from processor memory. It then
   immediately prefetches the next line from processor memory into its prefetch buffer.

Two problems exist. The first case is when the PCI bridge chip has requested the cache line at the end of a
physical page (note that this does not imply that the PCI device, itself, has requested the cache line at the end
of the page), e.g., 0x0fe0. The PCI bridge chip, incorrectly requests this cache line with the prefetch hint
enabled. The I/O bridge chip, to avoid fetching onto a possible non-existent page, but needing to do something
with the prefetch hint enabled, prefetches the first cache line of the page, e.g., 0x0000. In the example case,
stale data can be read if the next request from the PCI master is for address 0x0000, which has just been
incorrectly prefetched.

This case is fairly easy to hit. A driver might have control information consisting of a list of multiple
structures that just fill a physical page. If the PCI bridge reads the last cache line of the page, followed by the
driver re-writing the list, doing a dma_sync(), and then directing the PCI master to re-read the list, the PCI
master will read stale data in the first cache line.
The second case is where a driver has two adjacent data structures on the same page. The PCI master reads from the first data structure. The PCI bridge chip and the I/O bridge chip have prefetched such that the I/O bridge chip has a cache line in its prefetch buffer that actually resides in the second data structure. If the PCI master then reads that particular address, it may have stale data (depending upon the sequence the driver follows in updating it vis-a-vis the PCI master's access).

In both cases, if the mapping is done using `wsio_map()` with `IO_NO_SEQ` and `IO_SAFE` flag bits set, no problem exists (because the I/O bridge chip ignores the prefetch hint when the I/O TLB is set up by WSIO mapping services with this mapping).

Note that the `IO_NO_SEQ` and `IO_SAFE` flag bits will degrade MRM and MRL performance by about a factor of two for the page(s) in this type of mapping. If it is absolutely necessary, for performance reasons, to use `wsio_fastmap()` or `wsio_map()` without the `IO_NO_SEQ` and `IO_SAFE` flag bits set the coherency problem can be prevented by having the PCI master read a different address which will reset the I/O bridge chip's prefetch buffer. A read of any address using MRM, MRL or a normal read transaction by the PCI master will accomplish this, so you might have the PCI master re-read the previous cache line and then throw it away.

SEE ALSO

`wsio_map` (WSIO3), `wsio_fastmap` (WSIO3), `pci_errata-2`
NAME

 PCI_ERRATA-2 (PCI5) –

PCI TRANSACTION ORDERING

Due to interaction between the host bus, PCI bridge chips, and the PCI bus, in certain situations, the
Producer Consumer model requirements defined in the PCI 2.1 Specification may not be met. For more
detailed information refer to the discussion on Transaction Ordering in the PCI Chapter of the HP-UX Driver
Development Guide.

SEE ALSO

pci_errata-1 (PCI5)
NAME

PCI_ERRATA-3 (PCI5) –

PCI CONFIGURATION CYCLE RETRY PROBLEM

The system’s PCI bridge chip holds IRDY too long on config write retry. This problem has only been seen in simulation with some revisions of the PCI bridge chip used on the B1000, C3000, J5000, and the N-Class servers.

This problem occurs when:

1. A device retries a configuration cycle, and if
2. the device asserts DEVSEL and STOP during the same cycle, and if
3. that cycle is not the one immediately following the address cycle then the PCI bridge chip will ignore the RETRY, believe that the card never asserted DEVSEL response, which will cause a master abort.

If the card asserts DEVSEL for one or more cycles before it asserts STOP, the problem does not occur.

In PCI bridge chips exhibiting this behavior, the most likely result is an HPMC or panic.