This chapter presents routines and conceptual material specifically for drivers of Peripheral Component Interconnect (PCI) devices. PCI is an industry-standard bus supported as a bus-nexus CDIO on HP-UX systems as of Release 10.20, as a means of providing expansion I/O. PCI Services are a supplement to the WSIO HP-UX driver environment, providing PCI-specific functionality to drivers that use a PCI bus either as a means to providing expansion slots or for core I/O functionality.

In conjunction with the WSIO Services driver environment, PCI Services form the complete environment necessary to write an HP-UX driver capable of handling a PCI card. The services are generic in nature and not tied to any particular PCI bus adapter.

This chapter corresponds to the *PCI Local Bus Specification, Revision 2.1*. It also specifies the features, possible limitations, and assumptions of the services that you may need to be aware of.

The HP-UX PCI Services routines are described in the *HP-UX Driver Development Reference*; they are also summarized in the “PCI Services Summary” section.

**NOTE**  
The examples in this chapter follow the routine naming conventions described in Chapter 4, “Writing a Driver.”
PCI Overview

This section gives you a brief overview of PCI. It is not intended to be sufficient PCI information in itself. The developer should be familiar with the *PCI Local Bus Specification, Revision 2.1* before trying to write a driver for a PCI card.

PCI Register Spaces

There are three register spaces in PCI:

- PCI Configuration Space
- PCI Memory Space
- PCI I/O Space.

Generic configuration registers are placed in configuration space. Registers for card-specific control and status and for on-card data buffers are generally located in PCI memory space or (less often) in PCI I/O space.

PCI Configuration Space

PCI configuration space holds specific registers having to do with initialization and configuration of PCI devices. Some or all of this register space is the same for all PCI devices, giving generic initialization software the ability to recognize and configure all PCI-compliant devices.

This space is accessed primarily at startup time, when initialization occurs, but there is no prohibition on accessing it at other times after startup.

The following PCI Services access registers in PCI configuration space.

- pci_read_cfg_uintN_isc()
- pci_write_cfg_uintN_isc()

These functions take a configuration space offset (0x00–0xff) as their address inputs. See “Defined Constants”. The registers at addresses 0x00–0x3f are defined in the *PCI Local Bus Specification, Revision 2.1*, but the remainder of the space can be used by the card maker for any card-specific registers it sees fit to put there. In most cases, however, card-specific registers are placed in PCI memory space or PCI I/O space instead.

PCI Memory Space

Most cards place their registers for control, data buffering, and status in PCI memory space. In HP-UX systems, accesses to PCI memory space have higher performance than access to PCI I/O space. Registers mapped in PCI memory space respond to memory cycles on the PCI bus.

The following PCI Services access registers in PCI memory space.

- READ_REG_UINTn_ISC()
- WRITE_REG_UINTn_ISC()

These macros take Virtual Addresses (which are mapped to PCI memory addresses) as their address inputs. They have different effects depending on whether or not PCI_LITTLE_ENDIAN_ONLY is defined by the driver prior to including `pci.h`. See “The PCI_LITTLE_ENDIAN_ONLY Flag” for more details.

Mapped PCI memory space, on workstations only, can also be accessed directly. In this case the developer will have to handle “endian” issues oneself.
**PCI I/O Space**

Some cards place their registers for control, data buffering, and status in PCI I/O space. Registers mapped in PCI I/O space respond ONLY to I/O cycles on the PCI bus.

The following PCI Services access registers in PCI I/O space.

- `pci_read_port_uintN_isc()`
- `pci_write_port_uintN_isc()`

These functions take port handles and offsets as their address inputs.

**PCI Transaction Ordering**

This section covers the ordering of transactions to and from PCI space. These transactions include:

- Processor mastered reads and writes to PCI space
- PCI card mastered reads and writes to host memory
- Interleaved processor and PCI card mastered reads and writes of host memory space

Host bus to PCI bridges used in HP-UX systems need to comply with the transaction ordering requirements of both buses. As a result, in certain cases the order of completion guaranteed under the Producer-Consumer model as defined in the *PCI Local Bus Specification, Revision 2.1* is not met.

**Ordering of Processor Mastered Reads and Writes To PCI Space**

This section details transaction ordering for processor mastered PCI transactions. Typical examples of this type of transaction are reading and writing of registers on a PCI interface card.

**Blocking versus Nonblocking Transactions**

Processor mastered reads of PCI space are blocking transactions. This means that ordering is not a problem with reads, since only one read can occur at a time. A read holds the caller (processor) until it completes. The hardware implementation prevents a second processor reading from the same PCI space until the first processor's read completes.

Writes to PCI registers, on the other hand, are nonblocking (“posted”) transactions. This means that, to get better performance, the writing process does not wait for a write to complete after calling for it (writes do not block). The write will complete on its own, and the writer can do other things, including other writes, in the meantime. Because multiple outstanding uncompleted writes are possible (and common) under this model, ordering must be established on the completion of the writes.

Processor mastered PCI write ordering is relatively simple. If a processor writes to registers A, B, and C in that order, the writes will complete such that they are only observable in the same order (for example, you could never observe that B had been written but A had not been yet). If two or more processors are writing to registers, their ordering with respect to each other is considered irrelevant, but the ordering of their individual writes is preserved as above. This is the order of completion guaranteed under the Producer-Consumer model as defined in the *PCI Local Bus Specification, Revision 2.1*.

**Write Side-Effects**

The side-effects of any write are not guaranteed to happen immediately. Writes are posted; they will complete eventually.

All posted writes must be flushed and completed before any read is allowed to complete. So, to assume a write's effects have actually occurred, a read must be performed to flush the writes posted in the queue. You must keep this in mind when coding register writes; most of the time, it is acceptable to not know when a register write completes, but in some cases, you have to be careful.
A good example of such a case is when a driver’s interrupt service routine (ISR) is dealing with the interrupt request register (IRR) on a card. Clearing a bit in the register indicates that the interrupt has been serviced. This is done by posting a write to the register. If the driver posts this write and exits its ISR, it could conceivably get interrupted again immediately because the write hadn’t yet reached the bit in the IRR to tell it to stop trying to interrupt. One solution to this potential problem is to make sure to read back the value in the IRR before exiting from the ISR. Most drivers do this anyway so they can handle multiple interrupts in the same ISR visit.

Ordering of PCI Card Mastered Reads and Writes to Host Memory

We use the terms DMA read for a PCI mastered read from host memory, and DMA write for a PCI mastered write to host memory. In current hardware implementations, transaction ordering of DMA reads and DMA writes are ONLY preserved when the target memory location is contained in the same processor cacheline. In other cases, DMA reads are allowed to pass DMA writes and driver writers need to take this behavior into account.

If your driver needs the exact PCI producer-consumer behavior, as seen from the PCI card, you must ensure that the element(s) residing in host memory, requiring strict ordering, are physically on the same cacheline. Current hardware implementations have cachelines that are multiples of 32 bytes in length. For safety you should make sure that you limit your flag or status elements to 32 bytes aligned on MAX_CACHLINE_SIZE boundaries (defined in /usr/include/sys/dma.h).

Ordering of Interleaved Processor and PCI Card Mastered Reads and Writes to Host Memory

If your driver expects PCI or PA ordering rules to apply in this situation, you need to ensure that your producer-consumer elements reside on the same cacheline. The following scenario does not meet the producer-consumer transaction ordering requirements.

- Cacheline X holds the card’s status — initially “working”
- Cacheline Y holds the card’s next command — initially “go to sleep”
- Card finishes work and sets status in cacheline X to “done”
- Card reads its next command from cacheline Y
- Processor writes command to cacheline Y “do more work”
- Processor checks status in cacheline X

If the processor’s read of cacheline X (status) returns “working”, the processor assumes that the card has not checked its command yet. Therefore it has not gone to sleep and does not need to be awakened. If the status read returns “done”, the processor wakes up the card.

The crux of the problem here is that ordering is not enforced between the two cachelines and DMA reads can pass DMA writes. Thus, both the processor and the card’s reads can return the original value. This would result in the card going to sleep and the processor not waking it up.

If you cannot place the status and commands on the same cacheline, you must use some other means to ensure correct behavior. One possible workaround would be to set a timeout to ensure that the above deadlock did not occur. In most cases, commands are written to the card’s register, i.e., the command is not in host memory and the above scenario would not apply.
Ordering of Interleaved Processor Mastered Writes, PCI Card Mastered Reads of Host Memory, and Processor Mastered Writes of Host Memory

The following scenario does not meet the producer-consumer transaction ordering requirements.

- Processor writes a command to the PCI card to stop processing a task list in host memory, because the processor is about to update or change the list.
- Processor begins updating the task list in main memory.
- Card does a DMA read of the next (possibly being updated) element of the task list in main memory as a part of normal processing.
- Posted processor write to the card arrives at the card telling it to stop processing the list, which unfortunately it has just done.

The problem in this case is similar to the previous problem. DMA reads by a PCI master are allowed to pass processor writes to PCI space. Since processor writes are posted, ordering is not guaranteed on the combination of the internal system bus and the PCI bus. This situation can be avoided by doing a processor read of PCI space immediately following the processor write, as shown next.

- Processor writes a command to the PCI card to stop processing the task list.
- Processor does a “dummy” read of the PCI card (perhaps a status register on the card) to make sure that the “posted” write to PCI space has completed. Perhaps a read of card status is required here in any case to ensure that the DMA engine has stopped fetching tasks.
- Processor updates the task list in main memory.
- Processor writes a command to the PCI card to resume task processing.

The above behavior can occur on all shipping PCI based systems as of the date of this document. Drivers written for workstations should probably always ensure that, where necessary, posted writes are followed by “dummy” reads to ensure ordering. This behavior will probably not occur in servers due to chipset implementation, and may not occur in future workstation products.

PCI Endian Issues

HP PA-RISC is a big-endian system; for a multibyte quantity, the most significant byte (MSB) has the lowest address, and the least significant byte (LSB) has the highest. Intel’s i86 processors, on the other hand, are little-endian. Because PCI was derived from the PC world, it, too, is little-endian.

When multibyte words are transferred between the PCI bus and the system bus (HP PA-RISC), the bytes of the word are reversed or swapped by the hardware.

This insures that the receiving system can properly interpret and store the data, from most significant byte to least significant byte. This will not happen when the data is transferred byte-by-byte, but this method is inefficient.

Byte Swapping

In order for each system gets data in the format it expects, the PCI hardware uses a hardwired swapping mechanism at the interface between the two systems. The hardware swaps each byte of a 32-bit word so that all the bytes end up in the correct order on both sides of the interface. This means that large arrays of bytes, such as LAN packets and disk blocks, are in the correct order, even if they are transferred a multibyte word at a time.

This byte-ordering ensures that devices like disks, that are connected to the built-in SCSI on the internal system bus, can instead be connected to a SCSI card on the PCI bus.
When Pre-swapping is Required

Because of the byte swapping, the interpretation of multibyte integers is problematical. To see why this is so, assume that the transfer is occurring from the big-endian system to the little-endian system, and that swapping is being performed. If the byte array in question is a four-byte integer, it will be stored in big-endian format, MSB at the lowest address, on the little-endian side. If a device on the little-endian side of the interface decides to interpret these bytes as a four-byte integer; however, the “value” it will see will have all the bytes reversed. The same thing happens when transfers go in the opposite direction.

To correct the misinterpretation of multibyte integers on the opposite side of the bus, any multibyte quantity that is to be interpreted as an integer will have to be preswapped. This preswapping is then reversed by the hardwired swapping, making the value correct for integer interpretation on the other side of the interface. If the integer is stored in memory, however, it will end up reversed.

Several macros are provided in the file `pci.h` to assist in swapping data.
PCI Device Setup

This section is a collection of several pieces of information that you need to understand before attempting to set up a PCI device.

Mapping Base Address Registers into PCI Memory and I/O Space

When an HP-UX system boots, processor dependent code (PDC), I/O dependent code (IODC), and HP-UX system code maps a PCI card’s memory space base address registers into PCI memory space and I/O space base address registers into PCI I/O space.

The system attempts to map in all memory and I/O regions described by every PCI device or function’s memory and I/O base registers located in the PCI configuration space. If a suitable mapping is found, the system will write the base of the range back into the corresponding base address register. This address will be a PCI memory address if the base register identifies itself as a memory base, and a PCI I/O address if the base address identifies itself as an I/O base.

A driver's driver_attach() routine can then access the values loaded into the base registers in configuration space. It is important that a driver does not overwrite these addresses with different values, except as follows: As long as response to memory or I/O accesses via the command register has not yet been enabled, it is acceptable to store the register contents, write all ones to the register to determine the region size (as explained in the PCI Local Bus Specification, Revision 2.1) and, then restore the original contents.

Using the Base Address Registers

Before a driver can actually use these base addresses, another kind of “mapping” must take place. The problem is that the addresses placed in the base address registers by the system do not contain Virtual Addresses usable by the computer. Instead, they contain PCI addresses, used to talk on the bus. If a base address register is a memory base, it contains a PCI memory address. If it is an I/O base, it contains a PCI I/O address (See “PCI Register Spaces” for more information).

In either case, to use the PCI address in the base register, a mapping to a PA resource must take place, in order to allow the system to access the registers pointed to by the base.

It is very important that you do NOT arbitrarily mask bit 0 of a base address register. This bit indicates whether or not this particular set of registers responds to PCI memory cycles or PCI I/O cycles. During early PDC/IODC configuration, the defined base address registers are written in a manner prescribed by the PCI specification to determine size, alignment, and access type. If bit 0 is a “1” then PDC/IODC probing has determined that this particular register set ONLY responds to I/O cycles. If the base address register responds to I/O cycles, you MUST use the PCI services provided port I/O routines for access.

Using PCI Memory Base Registers

To use a PCI memory base address register, the range of PCI memory space must be mapped to a range of PA memory space. This is accomplished by calling map_mem_to_host(). The map_mem_to_host() call takes the PCI memory address (obtained directly from the base address register) and a size as inputs, and returns a Virtual Address that can be used to access that PCI address range. The accessor macros, READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC(), take PA virtual memory addresses as arguments, not PCI memory addresses.
NOTE
After reading a PCI memory base address register's value out of PCI configuration space, it is usually necessary to mask off the bottom four bits prior to making services calls such as map_mem_to_host(), since they have special values defined by the PCI Local Bus Specification, Revision 2.1 (See “Sample driver_attach() Routine” for an example).

Once this virtual mapping is done, the machine uses PA memory-mapped I/O to access the range. In other words, accesses to that range of PA memory space will be transmitted through into the PCI memory space. This just means that loads and stores to these PA memory addresses will result in loads and stores to the registers you wish to access.

For WSIO drivers, the if_reg_ptr member of the ISC structure is a Virtual Address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system. If if_reg_ptr is NULL, the driver needs to map the range itself (see “Mapping the Memory Base Register” for information on this). This is generally done in a PCI device's driver_attach() routine.

Using PCI I/O Base Registers
To use a PCI I/O base address register, the corresponding range of PCI I/O space must be mapped to a resource managed by PCI Services called a port handle, defined by the PCI structure PCI_PORT_HNDL. The PCI I/O space accessor functions, pci_read_port_uintN_isc() and pci_write_port_uintN_isc(), take port handles as arguments.

To do this mapping from a PCI I/O address to a port handle, the driver must read the I/O base registers from configuration space and call pci_get_port_handle_isc(), which takes a PCI I/O space address and a size as input, and returns a PCI_PORT_HNDL as output. This port handle (with an offset) is then used to access the registers in PCI I/O space.

NOTE
When reading a PCI I/O base address register's value out of PCI configuration space, it is necessary to mask off the bottom two bits prior to making a call to services such as pci_get_port_handle(), since they have special values defined by the PCI Local Bus Specification, Revision 2.1 (See pci_get_port_hndl_isc (PCI3) for an example).

Automatic IRQ Determination
PCI drivers calling isrlink() and isrunlink() should always pass -1 as the irq_line argument. This argument value causes the functions to read the needed IRQ information from the PCI device or function configuration space Interrupt Pin and/or Interrupt Line registers and use it to set up the ISR properly. If you need the IRQ information, you can read it from the Interrupt Line register.

Mapping the Memory Base Register
Many cards will have only a single range of registers (only a single memory base address register). For cards like these, the if_reg_ptr field in the ISC structure is useful.

PCI Services automatically maps one memory-space register into the isc->if_reg_ptr field in the following manner and with the following limitations:

- Only the first nonzero 32-bit memory base register found is mapped, starting at 0x10 and searching up to 0x24 inclusive. These are the six defined base address register locations in PCI configuration space. A Virtual Address for accessing this register is stored in if_reg_ptr.
However, if that base register’s size (the size of the register range) is in excess of 8 KB, it is NOT mapped and _if_reg_ptr_ is set to NULL. In this case, the driver itself must map the base registers it wants, using the PCI bus-dependent configuration access routines in conjunction with map_mem_to_host().

If _if_reg_ptr_ is NULL and the result of a map_mem_to_host() call is NULL, then for whatever reason, this particular address could NOT be mapped and you MUST NOT attempt to access it. These limitations are necessary to define which of many possible base registers will be mapped, as well as to prevent unnecessary use of translation lookaside buffers (TLB). If PCI Services do not map in any memory base register, or if there are more registers than the first one found as above, the driver can read the base registers explicitly from the PCI device or function’s configuration space and get a PA virtual mapping with the map_mem_to_host() kernel routine. (See “Sample driver_attach() Routine” for an example).

The limitations also prevent wasting of kernel resources on base registers that we may not wish to map in the normal way (for example, a graphics card frame buffer is an enormous range that should be treated differently from a regular register range). PCI Services has arbitrarily decreed that anything bigger than 8 KB should be dealt with by the driver, not mapped automatically by WSIO services.

**PCI Configuration Space Restrictions**

The registers in the PCI configuration space of each device are described in the *PCI Local Bus Specification, Revision 2.1*. Many of these registers are writable, but not every writable register is appropriate for a driver to modify. Some of the fields are set up on behalf of the driver and card by the system, which has information that a driver or card could not know about system parameters.

The basic guideline is that things that you do not understand or need not have anything to do with should not be altered. The following are some examples of configuration registers to leave alone:

- **The Command Register (most parts of it)**
  The command register must be written by drivers in order to enable bus-mastering, memory space access, and I/O space access, among other things. Many bits in this register are irrelevant to a driver and some have already been set by the system. Bits in the command register that may have been previously set must not be overwriten. Therefore, when a driver wants to set a bit in the register, it must first read the current state of the register, use bitwise OR or AND to make any changes, then write the value back. This procedure preserves bits previously set by the system.

- **The Latency Timer Register**
  This is set by the system. It should not be tampered with by individual drivers, as incorrect settings can degrade overall system performance.

- **The Cache Line Size Register**
  This register is set by the system to match the machine’s cache line. Drivers do not know the cache line size for the particular machine they are currently running on, so they should not change this register’s contents.

- **The Base Address Registers**
  The system uses the information in these registers to map their ranges into PCI memory and I/O space. It then writes a value back into the register corresponding to the base of the range it allocated. These ranges should not be overwritten by drivers, with one exception. In some cases, it may be necessary for a driver to determine the size and alignment of the range a base address register is mapped to. The procedure for getting this information involves writing all ones to the register, reading the result back, and decoding it for the needed values, as described in the *PCI Local Bus Specification, Revision 2.1*. Doing this is permitted as long as the original value is read and stored first, then restored to the register after the size has been determined. This should only be done before memory or I/O transactions to the card have been enabled through the command register.
The Interrupt Line Register

System-specific interrupt routing information is stored in this register. Writing a new value to it will probably cause the card to stop working.
PCI Device Operation

The PCI_LITTLE_ENDIAN_ONLY Flag

We recommended that drivers define the PCI_LITTLE_ENDIAN_ONLY flag before they include pci.h. This will help them get better performance from their I/O accesses.

Most PCI drivers are written for cards whose primary method of accessing registers is through PCI memory space.

PCI drivers written for workstations only, currently all third party drivers, may use direct C code constructs to access registers in PCI memory space. For example:

```c
myClearRegs(regsToInit,size)
int *regsToInit;
{int i;
 for (i=0;i<size;i++)
  *regsToInit++ = 0;
}
```

These drivers may also use the READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros with the PCI_LITTLE_ENDIAN_ONLY flag defined in the pci.h header. The choice of whether to directly access a register or to use one of the macros essentially becomes whether or not you want to swap.

The READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros are the safest accessors of PCI memory space, but what they are actually defined to do depends on whether or not the PCI_LITTLE_ENDIAN_ONLY flag was defined by the driver before the driver source code included the pci.h header.

If the driver does NOT explicitly define PCI_LITTLE_ENDIAN_ONLY before including pci.h, then the macros expand into function calls that are guaranteed to byte-swap correctly and perform the memory access. This can be considered “extra safe” mode; it will always work on all bus adapters in the future. The function calls guarantee PCI-adapter-independence. However, extra function call overhead is added to the register access, reducing its performance.

If the driver DOES explicitly define PCI_LITTLE_ENDIAN_ONLY, the performance loss due to the function call is taken away. In this case, the macros are expanded by the preprocessor into a series of in-line instructions that byte-swap and perform the access without a function call, under the assumption that the PCI adapter under which the card is running has directly mapped the PCI memory space into driver-accessible PA I/O space. This assumption is valid for all current and planned PCI adapters, with the single exception of a few special PA internal system bus based server PCI card projects. All regular drivers (i.e., those that are not explicitly written to drive a specially-equipped PA internal system bus based card) will benefit from defining the PCI_LITTLE_ENDIAN_ONLY flag and should do so before including pci.h.
The following pseudocode (resembling and summarizing the actual code in pci.h) may help explain the flag’s relation to the macros, and how and why to use it:

```c
#ifdef PCI_LITTLE_ENDIAN_ONLY
#define READ_REG_UINTn_ISC(isc, addr, value)  
    (*value = ENDIAN_SWAP_MACRO(*addr))
#define WRITE_REG_UINTn_ISC(isc, addr, value)  
    (*addr = ENDIAN_SWAP_MACRO(value))
#else /* *NOT* PCI_LITTLE_ENDIAN_ONLY */
#define READ_REG_UINTn_ISC(isc, addr, value)  
    isc->adapter_dependent_readN_function_call(addr, value)
#define WRITE_REG_UINTn_ISC(isc, addr, value)  
    sc->adapter_dependent_writeN_function_call(addr, value)
#endif /* PCI_LITTLE_ENDIAN_ONLY */
```

## Direct Memory Access (DMA)

A PCI device acting as a PCI bus master uses direct memory access (DMA) to generate read or write cycles that access locations in PA memory and card memory. DMA is a primary method of getting information to or from a card in large chunks, as opposed to doing many reads or writes to buffers of card registers.

PCI has no special routines to perform DMA. It uses the standard WSIO Services calls for bus-independent DMA, including:

- `init_map_context()`
- `wsio_map()`, `wsio_fastmap()`, and `wsio_unmap()`
- `dma_setup()` and `dma_cleanup()`
- The `iovec` and `dmaParms` structures

In the HP-UX Driver Development Reference, see `dma_cleanup` (WSIO3), `dmaParms` (WSIO4), `dma_setup` (WSIO3), `init_map_context` (CDIO3), `iovec` (KER4), `wsio_fastmap` (WSIO3), `wsio_map` (WSIO3), and `wsio_unmap` (WSIO3).

Be aware that certain combinations of WSIO mapping service calls can interact with PCI masters to create an inconsistent view of memory. See “PCI Masters and Coherence”.

Many EISA drivers make calls to functions like `eisa_dma_setup()` and `eisa_dma_cleanup()`. There are no corresponding PCI functions.

The only thing PCI-specific about performing DMA with a PCI device is that the device’s command register (`PCI_CS_COMMAND`) in PCI configuration space contains a bit (`PCI_CMD_BUS_MASTER`) that must be set (with `pci_write_cfg_uintN_isc()`) in order to allow the device to master the bus. The use of this bit is illustrated in “Sample driver_attach() Routine”.

---

**Chapter 14**
PCI Masters and Coherence

It is possible for prefetching of host memory by the hardware chipset to result in a PCI master reading stale data, even though the proper \texttt{dma_sync} calls have been made. This does not occur if the mapping is done with \texttt{wsio_map()} with flags \texttt{IO_NO_SEQ} and \texttt{IO_SAFE} set. See \texttt{pci_errata} (PCI5) in the \textit{HP-UX Driver Development Reference} for details.
Leveraging Existing Drivers

Multibus Drivers

Some cards for different buses have similar chip sets, making the programming models very similar for the base functionality. Consequently, a single driver can handle the functionality for the different bus cards.

Writing a Multibus Driver

A multibus driver is one in which a similar chip set appears on cards that plug into multiple buses. An example of this is the current SCSI driver. Similar SCSI chips exist in devices on the GSC bus, the EISA bus, and the PCI bus. A single driver, scsi_c720, is capable of controlling these SCSI chips no matter where they live.

Because the programming model of the base functionality is so similar, it makes sense to have a single driver to handle this functionality. Conversely, however, the bus-specific initialization of the nonbase functionality can often be radically different. The WSIO environment supports multibus drivers in the following ways:

- Many of the initialization functions are embedded in bus-independent functions that have bus-dependent implementations. This means that WSIO is responsible for making sure that the right thing is done when a driver calls a generic function like map_mem_to_host(). This moves the handling of bus-specific differences out of the driver and into the WSIO environment. Keeping the driver clean of calls specific to the current PCI adapter. See “Bus-Independent Functionality, Bus-Dependent Implementation”.

- Since each bus has a different attach chain, drivers can provide a separate driver_attach() routine for each bus. With careful handling, this can localize bus-specific functionality in the driver_attach() routines, allowing the driver_if_init() routine to handle bus-independent initialization and keeping the rest of the driver routines clean.

Whether or not you are planning to write a multibus driver, it is a good idea to keep as much PCI specifics in the driver_attach() routine as possible, just in case a card comes along someday for a new bus that uses the same or similar chips as the PCI card you are writing a driver for now. This is only a suggestion, as it does not make sense to compromise your current driver or make a huge and ungainly driver_attach() routine if there is clearly no need to.

Bus-Independent Functionality, Bus-Dependent Implementation

This class of functions allows multibus drivers to make a single call, allowing the driver environment to hide any bus-dependent implementation.

In PCI, the following features are supported. (There are a host of completely bus-independent functions that, by having no dependency on PCI, are supported by definition.)

- isrlink() and isrunlink(): Set irq_line to -1 to have the card supply the IRQ number to the system. See “Automatic IRQ Determination”.

- isc->if_reg_ptr value: One memory space base register is mapped automatically, subject to the conditions described in “Mapping the Memory Base Register”.

- wsio_map(), wsio_unmap(), wsio_dma_alloc(), wsio_dma_free(), and others in the WSIO family of coherent I/O DMA services.

- dma_setup() and dma_cleanup()

The WSIO functions wsio_get_interrupts() and wsio_get_registers() are not supported for PCI. See “Unsupported WSIO Functions” for details.
PCI Services Summary

PCI Services are accessed through special PCI functions that allow device and interface drivers to be much smaller and more supportable.

These functions are summarized here and described in detail in the *HP-UX Driver Development Reference*.

- **pci_desc_bus_transactions_isc()** — Allow a driver to describe the typical bus-performance-transaction size.
- **pci_get_fru_info_isc()** — Get the field replaceable unit (FRU) information for the device associated with an ISC.
- **pci_get_port_hndl_isc()** — Get a system-defined handle for manipulating the range of PCI I/O-space ports.
- **pci_read_cfg_uintN_isc()** — Read an 8-, 16-, or 32-bit unsigned integer from a PCI configuration register.
- **pci_read_port_uintN_isc()** — Read little-endian data from a PCI I/O-space port previously identified by a call to **pci_get_port_hndl_isc()**.
- **pci_unget_port_hndl_isc()** — Delete a handle returned by **pci_get_port_hndl_isc()**.
- **pci_write_cfg_uintN_isc()** — Write an 8-, 16-, or 32-bit unsigned integer into a PCI configuration register.
- **pci_write_port_uintN_isc()** — Write little-endian data to a PCI I/O port previously identified by a call to **pci_get_port_hndl_isc()**.
- **CONNECT_INIT_ROUTINE()** — Associate a *driver_if_init()* routine with the driver.
- **PCI_ATTACH_DEV_INIT_ERROR()** — Notify WSIO Services that an error occurred during a device’s initialization.
- **READ_REG_UINTn_ISC()** — Read and byte-swap 8-, 16-, or 32-bit data from a little-endian bus.
- **WRITE_REG_UINTn_ISC()** — Byte-swap and write 8-, 16-, or 32-bit data to a little-endian bus or a host memory area shared by the driver and a little-endian bus master.
Unsupported WSIO Functions

PCI Services do NOT support the following WSIO functions.

- **wsio_get_interrupts()**
  
  This function is provided to tell the driver what interrupt line a card is using. This information is primarily used to link or unlink an ISR. In PCI, `isrlink()` and `isrunlink()` should be called with `irq_line` set to -1, indicating that the system should determine the appropriate IRQ from the card, and rendering the `wsio_get_interrupts()` call unnecessary. If the IRQ is needed for some other reason, it can be read from the Interrupt Line register in PCI configuration space.

- **wsio_get_registers()**

  This function is designed to return the base address register for a card’s memory-mapped I/O. For PCI this information is available in the ISC at `driver_attach` time as the value `if_reg_ptr`. Also see “Mapping the Memory Base Register”.


Multiprocessor (MP) Safety

All PCI drivers should be coded to be MP safe.

Specifically, this means that they should not rely upon SPL levels to guarantee exclusive access to critical sections, but should instead protect their own critical sections using spinlocks, semaphores, and other methods of MP protection. See Chapter 3, "Multiprocessing," for details.
Constants and Data Structures

The constant definitions and data structures are defined in the PCI header file, `pci.h`.

User Visible PCI-Specific Data Structures

typedef struct _pci_id
{
    uint16_t vendor_id;
    uint16_t device_id;
} PCI_ID;

Defined Constants

/* Configuration space offsets. */
#define PCI_CS_VENDOR_ID 0x00
#define PCI_CS_DEVICE_ID 0x02
#define PCI_CS_COMMAND 0x04
#define PCI_CS_STATUS 0x06
#define PCI_CS_REV_ID 0x08
#define PCI_CS_CLASS_PROG_IF 0x09
#define PCI_CS_CLASS_SUB_CLASS 0x0a
#define PCI_CS_CLASS_BASE 0x0b
#define PCI_CS_CACHE_LINE_SIZE 0x0c
#define PCI_CS_LATENCY_TIMER 0x0d
#define PCI_CS_HEADER_TYPE 0x0e
#define PCI_CS_BIST 0x0f
#define PCI_CS_INTERRUPT_PIN 0x3d

/* masks for configuration data */
#define PCI_CS_MULT_FUNC_MASK 0x80

/* bit definitions for configuration space command register */
#define PCI_CMD_IO_SPACE 0x001
#define PCI_CMD_MEM_SPACE 0x002
#define PCI_CMD_BUS_MASTER 0x004
#define PCI_CMD_SPEC_CYCLES 0x008
#define PCI_CMD_MEM_WRINVAL_EN 0x010
#define PCI_CMD_VGA_PAL_SNOOP 0x020
#define PCI_CMD_PARITY_ERR_RESP 0x040
#define PCI_CMD_WAIT_CYCLE_CNTL 0x080
#define PCI_CMD_SERR_ENABLE 0x100
#define PCI_CMD_FAST_BACK_EN 0x200
A Sample PCI Driver

The following example is a skeleton that demonstrates how to write a PCI device driver in HP-UX using PCI and WSIO Services. The only part of this example that is PCI-specific is the driver_attach() routine. The other parts are typical of all WSIO drivers. They are included here for context and completeness. Chapter 4, “Writing a Driver,” contains more complete information on the structures and functions needed to write a WSIO driver.

We have a hypothetical PCI device, the ZZZ8109C PCI Blender card, for which we want to write a driver.

The blender is a character device, so our driver will be a character device driver. A character device is the counterpart of a block device, and has to do with how a device accesses its data and does DMA. The only type of PCI card that would be a block device would be a SCSI adapter or disk or tape drive controller.

Our example driver is written as a monolithic driver. This means it is both an interface driver (one that touches real hardware and registers) and a device driver (one that has a device special file). Even though we are writing both an interface and a device driver, we specify T_INTERFACE in the wsio_drv_info_t structure, since we cannot specify both.

Following the routine-naming conventions described in “Step 1: Choosing a Driver Name” in Chapter 4, “Writing a Driver,” we name the driver ZZZ and place it in the (arbitrary) class blender.

Sample WSIO Setup and Structures

We include the necessary header files. See the reference pages for each kernel call and data structure the driver uses to find out which headers the driver requires. WSIO drivers generally require the <wsio/wsio.h> header file. PCI drivers also require the <sys/pci.h> header file.

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

Next, we declare the driver’s routines that can be called by the kernel. These are used in the `drv_ops_t` structure.

```c
int ZZZ_open();
int ZZZ_close();
int ZZZ_read();
int ZZZ_write();
int ZZZ_ioctl();
```

We need a ZZZ_saved_attach function pointer to store the old head of the attach chain when we add our ZZZ_attach() routine to it in the ZZZ_install() routine. We also need values for vendor ID (ZZZ_VEN_ID) and device ID (ZZZ_DEV_ID) for the comparison in ZZZ_attach().

```c
static int (*ZZZ_saved_attach)();
int ZZZ_VEN_ID = value
/* these should be initialized */
int ZZZ_DEV_ID = value
/* these should be initialized */
```

The `drv_ops_t` structure specifies the “external” driver routines to the kernel. The flags specify that the driver should be called on all device closes and that it is MP safe. See the “The `drv_info_t` Structure Type” section in Chapter 4, “Writing a Driver,” for further details.

```c
static drv_ops_t ZZZ_ops =
{
    ZZZ_open, /* open */
    ZZZ_close, /* close */
    NULL, /* strategy */
```

The `drv_info_t` structure specifies the driver's name and class. The flags specify that the driver is character type and MP safe and that the configuration, including major number, should be saved and retained across reboots. See the “The `wsio_drv_data_t` Structure Type” section in Chapter 4, “Writing a Driver,” for further details.

```
static drv_info_t ZZZ_info =
{
    "ZZZ",            /* name */
    "blender",        /* class */
    DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE, /* flags */
    -1,               /* block major number (-1 for dynamic) */
    -1,               /* character major number (-1 for dynamic) */
    NULL,             /* reserved */
    NULL,             /* reserved */
    NULL,             /* reserved */
    C_ALLCLOSES | C_MGR_IS_MP /* flags */
};
```

The `wsio_drv_info_t` structure gives WSIO Services additional information about the driver. The entries specify the driver's interface type, that it is an interface (or monolithic) driver, and that it conforms to the Release 10.0 I/O specifications. See Chapter 4, “Writing a Driver,” for further details.

```
static wsio_drv_data_t ZZZ_data =
{
    "blender",        /* matches class name for T_INTERFACE drivers */
    T_INTERFACE,      /* drv_type - either T_DEVICE or T_INTERFACE */
    DRV_CONVERGED,    /* drv_flags */
    NULL,             /* optional function */
    NULL               /* optional function */
};
```

The `wsio_drv_info_t` structure ties the preceding three structures together into a single structure used in the `ZZZ_install()` routine's call to `wsio_install_driver()`. See Chapter 4, “Writing a Driver,” for further details.

```
static wsio_drv_info_t ZZZ_wsio_info =
{
    &ZZZ_info,
    &ZZZ_ops,
    &ZZZ_data
};
```
Sample WSIO Routines

Sample driver_install() Routine

A driver's `driver_install()` routine registers the driver and its structures with WSIO Services and the I/O subsystem. It also links the driver's `driver_attach()` function into the appropriate attach chain.

(If the device had a `driver_dev_init()` function, we would link it similarly into the `dev_init` chain here.)

```
NOTE
The name given to this routine is restricted. It must begin with the name of the driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” in Chapter 4, “Writing a Driver.”
```

```
int ZZZ_install(void)
{
    int ret;

    /*
    * Register our driver information with WSIO services
    */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret) {
        /*
        * If the install worked,
        * link ourselves into the pci_attach chain
        */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
    * Exit, returning the value we got
    * from the wsio_install_driver() call
    */
    return ret;
}
```

Sample driver_attach() Routine

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver's `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver put its `driver_attach()` function on the `pci_attach` chain, so the system calls it every time a new PCI device is found). The `driver_attach()` routine first checks to see if this is the type of hardware it can claim, then claims it if it wants it and performs whatever initialization the card requires.

- PCI Services will NOT enable a PCI device or function response to memory accesses, I/O accesses, or PCI device or function mastering of the bus. This ensures that a PCI device or function remains completely disconnected from the bus until after driver initialization. It is the responsibility of the driver to do this, as shown in the following sample attach routine.
The driver is responsible for ensuring that the contents of a memory or I/O base register are not zero. All zeros indicates that either the specified configuration space register is not implemented by the PCI device or function or that the system could not find the resources to map the corresponding space into the system. If alternate register mappings exist, and those base registers are not zero, it is acceptable for the driver to use those mappings instead.

The driver must enable memory access, I/O access, and DMA.

```c
int ZZZ_pci_attach(uint32_t parm, struct isc_table_type *isc)
{
    uint8_t rev_id;
    uint16_t command_reg;
    uint32_t base_addr;
    PCI_ID *id = (PCI_ID *)&parm /* for LP64 */
    
    /*
    * see if this is our card
    */
    if (!(id->vendor_id == ZZZ_VEN_ID &&
         id->device_id == ZZZ_DEV_ID) { 
        goto exit0;
    }
    
    /*
    * If we use a standard bus interface chip we need to
    * check subsystem vendor ID and subsystem ID here
    * to make sure that our driver should be the
    * driver claiming this device
    */
    
    /*
    * Get the card revision
    */
    pci_read_cfg_uint8_isc(isc, PCI_CS_REV_ID, &rev_id);
    
    /*
    * We must check the isc->if_reg_ptr
    * before we use it. If it's NULL,
    * we read our base register and map it ourselves.
    * But if isc->if_reg_ptr isn't NULL, PCI
    * services already did the mapping work for us
    */
    if (isc->if_reg_ptr == NULL) {
        /*
        * We need to map our own base address.
        * Save the value in if_reg_ptr.
        * Get our physical base memory address.
        * For ZZZ, memory is at reg 0x10
        */
        pci_read_cfg_uint32_isc(isc, 0x10, &base_addr);
        
        /*
        * make sure we have a memory BAR
        * instead of an IO BAR
        */
        if (base_addr & 0x01) {
            printf("ZZZ - no memory BAR\n");
            goto exit0;
        }
        
        /*
        * Mask off the bottom four bits of the PCI
        * memory base register (see PCI spec for
        */
    }
```
* significance)
 */
base_addr &= ~0xf;

/*
 * Ensure this base register was mapped in by the
 * system. If base_addr is 0, then the system
 * was unable to allocate us PCI memory space at all.
 */
if (base_addr == 0) {
    goto err0;
}

/*
 * Get a virtual translation for card registers.
 * Assume there are 512 bytes of registers.
 * Save the value in if_reg_ptr.
 */
if ((isc->if_reg_ptr = map_mem_to_host(isc, base_addr,
     512)) == NULL){
    goto err0;
}

/*
 * Use if_reg_ptr to access the registers.
 * Enable memory access and bus mastering
 * (note: other bits in the register must be preserved)
 */
pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&command_reg);
pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND,
    command_reg | PCI_CMD_MEM_SPACE | PCI_CMD_BUS_MASTER);

/*
 * Set up our interrupt handler.
 * Note that -1 is the third argument to isrlink().
 */
if (isrlink(isc, ZZZ_isr, -1, isc, 0) < 0) {
    goto err1;
}

/*
 * set up our init routine to be run later
 */
CONNECT_INIT_ROUTINE(isc, ZZZ_if_init);

/*
 * If everything okay, claim this card
 */
isc_claim(isc, &ZZZ_wsio_info);

/*
 * Exit without error
 */
goto exit0;

err1:

/*
 * clean up the mapping
 */
unmap_mem_from_host(isc, isc->if_reg_ptr, 512);

err0:

/*
 * indicate that we had an error
 */
PCI_ATTACHMENT_INIT_ERROR(isc);
exit0:
/*
 * Always exit by calling rest of chain
 * Use link established in ZZZ_install()
 */
    return ZZZ_saved_attach(parm, isc);
}

Other Driver Entry Point Routines

The other routines defined by the code above must also be declared and written. These functions include the following list:

- **ZZZ_if_init()**: Initialization of the card after the `driver_attach()` routine
- **ZZZ_isr()**: The driver's interrupt service routine
- **ZZZ_open()**: The `drv_ops_t`-defined entry point for `open()`.
- **ZZZ_close()**: The `drv_ops_t`-defined entry point for `close()`.
- **ZZZ_read()**: The `drv_ops_t`-defined entry point for `read()`.
- **ZZZ_write()**: The `drv_ops_t`-defined entry point for `write()`.
- **ZZZ_ioctl()**: The `drv_ops_t`-defined entry point for `ioctl()`.

The code for these functions is driver dependent. See “Step 6: Writing Entry Point Routines” and “Step 7: Writing Other Driver Routines” in Chapter 4, “Writing a Driver.” See also `close(2)`, `ioctl(2)`, `open(2)`, `read(2)`, `write(2)`.