PCI

|  |
| --- |
| **Contents**   [[hide](http://wiki.osdev.org/PCI)]   * [1 The PCI Bus](http://wiki.osdev.org/PCI#The_PCI_Bus) * [2 Configuration Space](http://wiki.osdev.org/PCI#Configuration_Space)   + [2.1 Configuration Mechanism #1](http://wiki.osdev.org/PCI#Configuration_Mechanism_.231)   + [2.2 PCI Device Structure](http://wiki.osdev.org/PCI#PCI_Device_Structure)   + [2.3 Base Address Registers](http://wiki.osdev.org/PCI#Base_Address_Registers)   + [2.4 Class Codes](http://wiki.osdev.org/PCI#Class_Codes) * [3 Enumerating PCI Buses](http://wiki.osdev.org/PCI#Enumerating_PCI_Buses)   + [3.1 "Brute Force" Scan](http://wiki.osdev.org/PCI#.22Brute_Force.22_Scan)   + [3.2 Recursive Scan](http://wiki.osdev.org/PCI#Recursive_Scan)   + [3.3 Recursive Scan With Bus Configuration](http://wiki.osdev.org/PCI#Recursive_Scan_With_Bus_Configuration) * [4 IRQ Handling](http://wiki.osdev.org/PCI#IRQ_Handling) * [5 Multifunction Devices](http://wiki.osdev.org/PCI#Multifunction_Devices) * [6 Disclaimer](http://wiki.osdev.org/PCI#Disclaimer) * [7 References](http://wiki.osdev.org/PCI#References) * [8 See Also](http://wiki.osdev.org/PCI#See_Also)   + [8.1 External Links](http://wiki.osdev.org/PCI#External_Links) |

The PCI Bus

The PCI bus was defined to establish a high performance and low cost local bus that would remain through several generations of products. By combining a transparent upgrade path from 132 MB/s (32-bit at 33 MHz) to 528 MB/s (64-bit at 66 MHz) and both 5 volt and 3.3 volt signaling environments, the PCI bus meets the needs of both low end desktop systems as well as that of high-end LAN servers. The PCI bus component and add-in card interface is processor independent, enabling an efficient transition to future processors, as well as use with multiple processor architectures. The disadvantage of the PCI bus is the limited number of electrical loads it can drive. A single PCI bus can drive a maximum of 10 loads. (Remember when counting the number of loads on the bus, a connector counts as one load and the PCI device counts as another, and sometimes two.)

Configuration Space

The PCI specification provides for totally software driven initialization and configuration of each device (or target) on the PCI Bus via a separate Configuration Address Space. All PCI devices, except host bus bridges, are required to provide 256 bytes of configuration registers for this purpose.

Configuration read/write cycles are used to access the Configuration Space of each target device. A target is selected during a configuration access when its IDSEL signal is asserted. The IDSEL acts as the classic "chip select" signal. During the address phase of the configuration cycle, the processor can address one of 64 32-bit registers within the configuration space by placing the required register number on address lines 2 through 7 (AD[7..2]) and the byte enable lines.

PCI devices are inherently little ENDIAN , meaning all multiple byte fields have the least significant values at the lower addresses. This requires a Big ENDIAN" processor, such as a Power PC, to perform the proper byte-swapping of data read from or written to the PCI device, including any accesses to the Configuration Address Space.

Systems must provide a mechanism that allows access to the PCI configuration space, as most CPUs do not have any such mechanism. This task is usually performed by the Host to PCI Bridge (Host Bridge). Two distinct mechanisms are defined to allow the software to generate the required configuration accesses. Configuration mechanism #1 is the preferred method, while mechanism #2 is provided for backward compatibility. Only configuration mechanism #1 will be described here, as it is the only access mechanism that will be used in the future.

**Configuration Mechanism #1**

Two 32-bit I/O locations are used, the first location (0xCF8) is named CONFIG\_ADDRESS, and the second (0xCFC) is called CONFIG\_DATA. CONFIG\_ADDRESS specifies the configuration address that is required to be accesses, while accesses to CONFIG\_DATA will actually generate the configuration access and will transfer the data to or from the CONFIG\_DATA register.

The CONFIG\_ADDRESS is a 32-bit register with the format shown in following figure. Bit 31 is an enable flag for determining when accesses to CONFIG\_DATA should be translated to configuration cycles. Bits 23 through 16 allow the configuration software to choose a specific PCI bus in the system. Bits 15 through 11 select the specific device on the PCI Bus. Bits 10 through 8 choose a specific function in a device (if the device supports multiple functions). Bits 7 through 2 select the specific 32-bit area in the device's configuration space.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **31** | **30 - 24** | **23 - 16** | **15 – 11** | **10 - 8** | **7 - 2** | **1 - 0** |
| Enable Bit | Reserved | Bus Number | Device Number | Function Number | Register Number | 00 |

The following code segment illustrates the use of configuration mechanism #1. Note that this segment, the functions sysOutLong and sysInLong are assembly language functions that make use of the OUTL and INL Pentium assembly language instructions.

unsigned short pciConfigReadWord (unsigned short bus, unsigned short slot,

unsigned short func, unsigned short offset)

{

unsigned long address;

unsigned long lbus = (unsigned long)bus;

unsigned long lslot = (unsigned long)slot;

unsigned long lfunc = (unsigned long)func;

unsigned short tmp = 0;

*/\* create configuration address as per Figure 1 \*/*

address = (unsigned long)((lbus << 16) | (lslot << 11) |

(lfunc << 8) | (offset & 0xfc) | ((UINT32)0x80000000));

*/\* write out the address \*/*

sysOutLong (0xCF8, address);

*/\* read in the data \*/*

*/\* (offset & 2) \* 8) = 0 will choose the fisrt word of the 32 bits register \*/*

tmp = (unsigned short)((sysInLong (0xCFC) >> ((offset & 2) \* 8)) & 0xffff);

return (tmp);

}

When a configuration access attempts to select a device that does not exist, the host bridge will complete the access without error, dropping all data on writes and returning all ones on reads. The following code segment illustrates the read of a non-existent device.

unsigned short pciCheckVendor(unsigned short bus, unsigned short slot)

{

unsigned short vendor,device;

*/\* try and read the first configuration register. Since there are no \*/*

*/\* vendors that == 0xFFFF, it must be a non-existent device. \*/*

if ((vendor = pciConfigReadWord(bus,slot,0,0)) != 0xFFFF) {

device = pciConfigReadWord(bus,slot,0,2);

. . .

} return (vendor);

}

**PCI Device Structure**

The PCI Specification defines the organization of the 256-byte Configuration Space registers and imposes a specific template for the space. Figures 2 & 3 show the layout of the 256-byte Configuration space. All PCI compliant devices must support the Vendor ID, Device ID, Command and Status, Revision ID, Class Code and Header Type fields. Implementation of the other registers is optional, depending upon the devices functionality.

The following field descriptions are common to all Header Types:

* **Device ID:** Identifies the particular device. Where valid IDs are allocated by the vendor.
* **Vendor ID:** Identifies the manufacturer of the device. Where valid IDs are allocated by PCI-SIG to ensure uniqueness and 0xFFFF is an invalid value that will be returned on read accesses to Configuration Space registers of non-existent devices.
* **Status:** A register used to record status information for PCI bus related events.
* **Command:** Provides control over a device's ability to generate and respond to PCI cycles. Where the only functionality guaranteed to be supported by all devices is, when a 0 is written to this register, the device is disconnected from the PCI bus for all accesses except Configuration Space access.
* **Class Code:** A read-only register that specifies the type of function the device performs.
* **Subclass:** A read-only register that specifies the specific function the device performs.
* **Prog IF:** A read-only register that specifies a register-level programming interface the device has, if it has any at all.
* **Revision ID:** Specifies a revision identifier for a particular device. Where valid IDs are allocated by the vendor.
* **BIST:** Represents that status and allows control of a devices BIST (built-in self test).
* **Header Type:** Identifies the layout of the rest of the header begining at byte 0x10 of the header and also specifies whether or not the device has multiple functions. Where a value of 0x00 specifies a general device, a value of 0x01 specifies a PCI-to-PCI bridge, and a value of 0x02 specifies a CardBus bridge. If bit 7 of this register is set, the device has multiple functions; otherwise, it is a single function device.
* **Latency Timer:** Specifies the latency timer in units of PCI bus clocks.
* **Cache Line Size:** Specifies the system cache line size in DWORD (32-bit) units. A device can limit the number of cacheline sizes it can support, if a unsupported value is written to this field, the device will behave as if a value of 0 was written.

This table is applicable if the Header Type is 00h. (Figure 2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **register** | **bits 31-24** | **bits 23-16** | **bits 15-8** | **bits 7-0** |
| **00** | Device ID | | Vendor ID | |
| **04** | Status | | Command | |
| **08** | Class code | Subclass | Prog IF | Revision ID |
| **0C** | BIST | Header type | Latency Timer | Cache Line Size |
| **10** | Base address #0 (BAR0) | | | |
| **14** | Base address #1 (BAR1) | | | |
| **18** | Base address #2 (BAR2) | | | |
| **1C** | Base address #3 (BAR3) | | | |
| **20** | Base address #4 (BAR4) | | | |
| **24** | Base address #5 (BAR5) | | | |
| **28** | Cardbus CIS Pointer | | | |
| **2C** | Subsystem ID | | Subsystem Vendor ID | |
| **30** | Expansion ROM base address | | | |
| **34** | Reserved | | | Capabilities Pointer |
| **38** | Reserved | | | |
| **3C** | Max latency | Min Grant | Interrupt PIN | Interrupt Line |

The following field descriptions apply if the Header Type is 0x00:

* **CardBus CIS Pointer:** Points to the Card Information Structure and is used by devices that share silicon between CardBus and PCI.
* **Interrupt Line:** Specifies which input of the system interrupt controllers the device's interrupt pin is connected to and is implemented by any device that makes use of an interrupt pin. For the x86 architecture this register corresponds to the PIC IRQ numbers 0-15 (and not I/O APIC IRQ numbers) and a value of 0xFF defines no connection.
* **Interrupt Pin:** Specifies which interrupt pin the device uses. Where a value of 0x01 is INTA#, 0x02 is INTB#, 0x03 is INTC#, 0x04 is INTD#, and 0x00 means the device does not use an interrupt pin.
* **Max Latency:** A read-only register that specifies how often the device needs access to the PCI bus (in 1/4 microsecond units).
* **Min Grant:** A read-only register that specifies the burst period length, in 1/4 microsecond units, that the device needs (assuming a 33 MHz clock rate).
* **Capabilities Pointer:** Points to a linked list of new capabilities implemented by the device. Used if bit 4 of the status register (Capabilities List bit) is set to 1. The bottom two bits are reserved and should be masked before the Pointer is used to access the Configuration Space.

This table is applicable if the Header Type is 01h (PCI-to-PCI bridge) (Figure 3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **register** | **bits 31-24** | **bits 23-16** | **bits 15-8** | **bits 7-0** |
| **00** | Device ID | | Vendor ID | |
| **04** | Status | | Command | |
| **08** | Class code | Subclass | Prog IF | Revision ID |
| **0C** | BIST | Header type | Latency Timer | Cache Line Size |
| **10** | Base address #0 (BAR0) | | | |
| **14** | Base address #1 (BAR1) | | | |
| **18** | Secondary Latency Timer | Subordinate Bus Number | Secondary Bus Number | Primary Bus Number |
| **1C** | Secondary Status | | I/O Limit | I/O Base |
| **20** | Memory Limit | | Memory Base | |
| **24** | Prefetchable Memory Limit | | Prefetchable Memory Base | |
| **28** | Prefetchable Base Upper 32 Bits | | | |
| **2C** | Prefetchable Limit Upper 32 Bits | | | |
| **30** | I/O Limit Upper 16 Bits | | I/O Base Upper 16 Bits | |
| **34** | Reserved | | | Capability Pointer |
| **38** | Expansion ROM base address | | | |
| **3C** | Bridge Control | | Interrupt PIN | Interrupt Line |

Here is the layout of the Header Type register:

|  |  |
| --- | --- |
| **Bit 7** | **Bits 6 to 0** |
| **MF** | Header Type |

**MF** - If MF = 1 Then this device has multiple functions.

**Header Type** - 00h Standard Header - 01h PCI-to-PCI Bridge - 02h CardBus Bridge

Here is the layout of the BIST register:

|  |  |  |  |
| --- | --- | --- | --- |
| **Bit 7** | **Bit 6** | **Bits 4 and 5** | **Bits 0 to 3** |
| BIST Capable | Start BIST | Reserved | Completion Code |

**BIST Capable** - Will return 1 the device supports BIST.

**Start BIST** - When set to 1 the BIST is invoked. This bit is reset when BIST completes. If BIST does not complete after 2 seconds the device should be failed by system software.

**Completion Code** - Will return 0, after BIST execution, if the test completed successfully.

This table is applicable if the Header Type is 02h (PCI-to-CardBus bridge)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **register** | **bits 31-24** | **bits 23-16** | **bits 15-8** | **bits 7-0** |
| **00** | Device ID | | Vendor ID | |
| **04** | Status | | Command | |
| **08** | Class code | Subclass | Prog IF | Revision ID |
| **0C** | BIST | Header type | Latency Timer | Cache Line Size |
| **10** | CardBus Socket/ExCa base address | | | |
| **14** | Secondary status | | Reserved | Offset of capabilities list |
| **18** | CardBus latency timer | Subordinate bus number | CardBus bus number | PCI bus number |
| **1C** | Memory Base Address 0 | | | |
| **20** | Memory Limit 0 | | | |
| **24** | Memory Base Address 1 | | | |
| **28** | Memory Limit 1 | | | |
| **2C** | I/O Base Address 0 | | | |
| **30** | I/O Limit 0 | | | |
| **34** | I/O Base Address 1 | | | |
| **38** | I/O Limit 1 | | | |
| **3C** | Bridge Control | | Interrupt PIN | Interrupt Line |
| **40** | Subsystem Vendor ID | | Subsystem Device ID | |
| **44** | 16-bit PC Card legacy mode base address | | | |

Here is the layout of the Command register:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bits 11 to 15** | **Bit 10** | **Bit 9** | **Bit 8** | **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bit 2** | **Bit 1** | **Bit 0** |
| Reserved | Interupt Disable | Fast Back-to-Back Enable | SERR# Enable | Reserved | Parity Error Response | VGA Palette Snoop | Memory Write and Invalidate Enable | Special Cycles | Bus Master | Memory Space | I/O Space |

**Interrupt Disable** - If set to 1 the assertion of the devices INTx# signal is disabled; otherwise, assertion of the signal is enabled.

**Fast Back-Back Enable** - If set to 1 indicates a device is allowed to generate fast back-to-back transactions; otherwise, fast back-to-back transactions are only allowed to the same agent.

**SERR# Enable** - If set to 1 the SERR# driver is enabled; otherwise, the driver is disabled.

**Bit 7** - As of revision 3.0 of the PCI local bus specification this bit is hardwired to 0. In earlier versions of the specification this bit was used by devices and may have been hardwired to 0, 1, or implemented as a read/write bit.

**Parity Error Response** - If set to 1 the device will take its normal action when a parity error is detected; otherwise, when an error is detected, the device will set bit 15 of the Status register (Detected Parity Error Status Bit), but will not assert the PERR# (Parity Error) pin and will continue operation as normal.

**VGA Palette Snoop** - If set to 1 the device does not respond to palette register writes and will snoop the data; otherwise, the device will trate palette write accesses like all other accesses.

**Memory Write and Invalidate Enable** - If set to 1 the device can generate the Memory Write and Invalidate command; otherwise, the Memory Write command must be used.

**Special Cycles** - If set to 1 the device can monitor Special Cycle operations; otherwise, the device will ignore them.

**Bus Master** - If set to 1 the device can behave as a bus master; otherwise, the device can not generate PCI accesses.

**Memory Space** - If set to 1 the device can respond to Memory Space accesses; otherwise, the device's response is disabled.

**I/O Space** - If set to 1 the device can respond to I/O Space accesses; otherwise, the device's response is disabled.

Here is the layout of the Status register:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit 15** | **Bit 14** | **Bit 13** | **Bit 12** | **Bit 11** | **Bits 9 and 10** | **Bit 8** | **Bit 7** | **Bit 6** | **Bit 5** | **Bit 4** | **Bit 3** | **Bits 0 to 2** |
| Detected Parity Error | Signaled System Error | Received Master Abort | Received Target Abort | Signaled Target Abort | DEVSEL Timing | Master Data Parity Error | Fast Back-to-Back Capable | Reserved | 66 MHz Capable | Capabilities List | Interrupt Status | Reserved |

**Detected Parity Error** - This bit will be set to 1 whenever the device detects a parity error, even if parity error handling is disabled.

**Signaled System Error** - This bit will be set to 1 whenever the device asserts SERR#.

**Received Master Abort** - This bit will be set to 1, by a master device, whenever its transaction (except for Special Cycle transactions) is terminated with Master-Abort.

**Received Target Abort** - This bit will be set to 1, by a master device, whenever its transaction is terminated with Target-Abort.

**Signaled Target Abort** - This bit will be set to 1 whenever a target device terminates a transaction with Target-Abort.

**DEVSEL Timing** - Read only bits that represent the slowest time that a device will assert DEVSEL# for any bus command except Configuration Space read and writes. Where a value of 0x00 represents fast timing, a value of 0x01 represents medium timing, and a value of 0x02 represents slow timing.

**Master Data Parity Error** - This bit is only set when the following conditions are met. The bus agent asserted PERR# on a read or observed an assertion of PERR# on a write, the agent setting the bit acted as the bus master for the operation in which the error occurred, and bit 6 of the Command register (Parity Error Response bit) is set to 1.

**Fast Back-to-Back Capable** - If set to 1 the device can accept fast back-to-back transactions that are not from the same agent; otherwise, transactions can only be accepted from the same agent.

**Bit 6** - As of revision 3.0 of the PCI local bus specification this bit is reserved. In revision 2.1 of the specification this bit was used to indicate whether or not a device supported User Definable Features.

**66 Mhz Capable** - If set to 1 the device is capable of running at 66 Mhz; otherwise, the device runs at 33 MHz.

**Capabilities List** - If set to 1 the device implements the pointer for a New Capabilities Linked list at offset 0x34; otherwise, the linked list is not available.

**Interrupt Status** - Represents the state of the device's INTx# signal. If set to 1 and bit 10 of the Command register (Interrupt Disable bit) is set to 0 the signal will be asserted; otherwise, the signal will be ignored.

Recall that the PCI devices follow little ENDIAN ordering. The lower addresses contain the least significant portions of the field. Software to manipulate this structure must take particular care that the endian-ordering follows the PCI devices, not the CPUs.

**Base Address Registers**

Base address Registers (or BARs) can be used to hold memory addresses used by the device, or offsets for port addresses. Typically, memory address BARs need to be located in physical ram while I/O space BARs can reside at any memory address (even beyond physical memory). To distinguish between them, you can check the value of the lowest bit. The following tables describe the two types of BARs:

|  |  |  |  |
| --- | --- | --- | --- |
| Memory Space BAR Layout | | | |
| **31 - 4** | **3** | **2 - 1** | **0** |
| 16-Byte Aligned Base Address | Prefetchable | Type | Always 0 |

|  |  |  |
| --- | --- | --- |
| I/O Space BAR Layout | | |
| **31 - 2** | **1** | **0** |
| 4-Byte Aligned Base Address | Reserved | Always 1 |

The Type field of the Memory Space BAR Layout specifies the size of the base register and where in memory it can be mapped. If it has a value of 0x00 then the base register is 32-bits wide and can be mapped anywhere in the 32-bit Memory Space. A value of 0x02 means the base register is 64-bits wide and can be mapped anywhere in the 64-bit Memory Space (A 64-bit base address register consumes 2 of the base address registers available). A value of 0x01 is reserved as of revision 3.0 of the PCI Local Bus Specification. In earlier versions it was used to support memory space below 1MB (16-bit wide base register that can be mapped anywhere in the 16-bit Memory Space).

When you want to retrieve the actual base address of a BAR, be sure to mask the lower bits. For 16-Bit Memory Space BARs, you calculate (BAR[x] & 0xFFF0). For 32-Bit Memory Space BARs, you calculate (BAR[x] & 0xFFFFFFF0). For 64-Bit Memory Space BARs, you calculate ((BAR[x] & 0xFFFFFFF0) + ((BAR[x+1] & 0xFFFFFFFF) << 32)) For I/O Space BARs, you calculate (BAR[x] & 0xFFFFFFFC).

To determine the amount of address space needed by a PCI device, you must save the original value of the BAR, write a value of all 1's to the register, then read it back. The amount of memory can then be determined by masking the information bits, performing a bitwise NOT ('~' in C), and incrementing the value by 1. The original value of the BAR should then be restored. The BAR register is naturally aligned and as such you can only modify the bits that are set. For example, if a device utilizes 16 MB it will have BAR0 filled with 0xFF000000 (0x01000000 after decoding) and you can only modify the upper 8-bits. [[1]](http://www.pcisig.com/reflector/msg05233.html)

**Class Codes**

The Class Code, Subclass, and Prog IF registers are used to identify the device's type, the device's function, and the device's register-level programming interface, respectively.

The following table represents the possible device types:

|  |  |
| --- | --- |
| **Class Code** | **Description** |
| 0x00 | Device was built prior definition of the class code field |
| 0x01 | Mass Storage Controller |
| 0x02 | Network Controller |
| 0x03 | Display Controller |
| 0x04 | Multimedia Controller |
| 0x05 | Memory Controller |
| 0x06 | Bridge Device |
| 0x07 | Simple Communication Controllers |
| 0x08 | Base System Peripherals |
| 0x09 | Input Devices |
| 0x0A | Docking Stations |
| 0x0B | Processors |
| 0x0C | Serial Bus Controllers |
| 0x0D | Wireless Controllers |
| 0x0E | Intelligent I/O Controllers |
| 0x0F | Satellite Communication Controllers |
| 0x10 | Encryption/Decryption Controllers |
| 0x11 | Data Acquisition and Signal Processing Controllers |
| 0x12 - 0xFE | Reserved |
| 0xFF | Device does not fit any defined class. |

The following table represents the possible device functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Class Code** | **Subclass** | **Prog IF** | **Description** |
| 0x00 | 0x00 | 0x00 | Any device except for VGA-Compatible devices |
| 0x01 | 0x00 | VGA-Compatible Device |
| 0x01 | 0x00 | 0x00 | SCSI Bus Controller |
| 0x01 | 0x-- | IDE Controller |
| 0x02 | 0x00 | Floppy Disk Controller |
| 0x03 | 0x00 | IPI Bus Controller |
| 0x04 | 0x00 | RAID Controller |
| 0x05 | 0x20 | ATA Controller (Single DMA) |
| 0x30 | ATA Controller (Chained DMA) |
| 0x06 | 0x00 | Serial ATA (Vendor Specific Interface) |
| 0x01 | Serial ATA (AHCI 1.0) |
| 0x07 | 0x00 | Serial Attached SCSI (SAS) |
| 0x80 | 0x00 | Other Mass Storage Controller |
| 0x02 | 0x00 | 0x00 | Ethernet Controller |
| 0x01 | 0x00 | Token Ring Controller |
| 0x02 | 0x00 | FDDI Controller |
| 0x03 | 0x00 | ATM Controller |
| 0x04 | 0x00 | ISDN Controller |
| 0x05 | 0x00 | WorldFip Controller |
| 0x06 | 0x-- | PICMG 2.14 Multi Computing |
| 0x80 | 0x00 | Other Network Controller |
| 0x03 | 0x00 | 0x00 | VGA-Compatible Controller |
| 0x01 | 8512-Compatible Controller |
| 0x01 | 0x00 | XGA Controller |
| 0x02 | 0x00 | 3D Controller (Not VGA-Compatible) |
| 0x80 | 0x00 | Other Display Controller |
| 0x04 | 0x00 | 0x00 | Video Device |
| 0x01 | 0x00 | Audio Device |
| 0x02 | 0x00 | Computer Telephony Device |
| 0x80 | 0x00 | Other Multimedia Device |
| 0x05 | 0x00 | 0x00 | RAM Controller |
| 0x01 | 0x00 | Flash Controller |
| 0x80 | 0x00 | Other Memory Controller |
| 0x06 | 0x00 | 0x00 | Host Bridge |
| 0x01 | 0x00 | ISA Bridge |
| 0x02 | 0x00 | EISA Bridge |
| 0x03 | 0x00 | MCA Bridge |
| 0x04 | 0x00 | PCI-to-PCI Bridge |
| 0x01 | PCI-to-PCI Bridge (Subtractive Decode) |
| 0x05 | 0x00 | PCMCIA Bridge |
| 0x06 | 0x00 | NuBus Bridge |
| 0x07 | 0x00 | CardBus Bridge |
| 0x08 | 0x-- | RACEway Bridge |
| 0x09 | 0x40 | PCI-to-PCI Bridge (Semi-Transparent, Primary) |
| 0x80 | PCI-to-PCI Bridge (Semi-Transparent, Secondary) |
| 0x0A | 0x00 | InfiniBrand-to-PCI Host Bridge |
| 0x80 | 0x00 | Other Bridge Device |
| 0x07 | 0x00 | 0x00 | Generic XT-Compatible Serial Controller |
| 0x01 | 16450-Compatible Serial Controller |
| 0x02 | 16550-Compatible Serial Controller |
| 0x03 | 16650-Compatible Serial Controller |
| 0x04 | 16750-Compatible Serial Controller |
| 0x05 | 16850-Compatible Serial Controller |
| 0x06 | 16950-Compatible Serial Controller |
| 0x01 | 0x00 | Parallel Port |
| 0x01 | Bi-Directional Parallel Port |
| 0x02 | ECP 1.X Compliant Parallel Port |
| 0x03 | IEEE 1284 Controller |
| 0xFE | IEEE 1284 Target Device |
| 0x02 | 0x00 | Multiport Serial Controller |
| 0x03 | 0x00 | Generic Modem |
| 0x01 | Hayes Compatible Modem (16450-Compatible Interface) |
| 0x02 | Hayes Compatible Modem (16550-Compatible Interface) |
| 0x03 | Hayes Compatible Modem (16650-Compatible Interface) |
| 0x04 | Hayes Compatible Modem (16750-Compatible Interface) |
| 0x04 | 0x00 | IEEE 488.1/2 (GPIB) Controller |
| 0x05 | 0x00 | Smart Card |
| 0x80 | 0x00 | Other Communications Device |
| 0x08 | 0x00 | 0x00 | Generic 8259 PIC |
| 0x01 | ISA PIC |
| 0x02 | EISA PIC |
| 0x10 | I/O APIC Interrupt Controller |
| 0x20 | I/O(x) APIC Interrupt Controller |
| 0x01 | 0x00 | Generic 8237 DMA Controller |
| 0x01 | ISA DMA Controller |
| 0x02 | EISA DMA Controller |
| 0x02 | 0x00 | Generic 8254 System Timer |
| 0x01 | ISA System Timer |
| 0x02 | EISA System Timer |
| 0x03 | 0x00 | Generic RTC Controller |
| 0x01 | ISA RTC Controller |
| 0x04 | 0x00 | Generic PCI Hot-Plug Controller |
| 0x80 | 0x00 | Other System Peripheral |
| 0x09 | 0x00 | 0x00 | Keyboard Controller |
| 0x01 | 0x00 | Digitizer |
| 0x02 | 0x00 | Mouse Controller |
| 0x03 | 0x00 | Scanner Controller |
| 0x04 | 0x00 | Gameport Controller (Generic) |
| 0x10 | Gameport Contrlller (Legacy) |
| 0x80 | 0x00 | Other Input Controller |
| 0x0A | 0x00 | 0x00 | Generic Docking Station |
| 0x80 | 0x00 | Other Docking Station |
| 0x0B | 0x00 | 0x00 | 386 Processor |
| 0x01 | 0x00 | 486 Processor |
| 0x02 | 0x00 | Pentium Processor |
| 0x10 | 0x00 | Alpha Processor |
| 0x20 | 0x00 | PowerPC Processor |
| 0x30 | 0x00 | MIPS Processor |
| 0x40 | 0x00 | Co-Processor |
| 0x0C | 0x00 | 0x00 | IEEE 1394 Controller (FireWire) |
| 0x10 | IEEE 1394 Controller (1394 OpenHCI Spec) |
| 0x01 | 0x00 | ACCESS.bus |
| 0x02 | 0x00 | SSA |
| 0x03 | 0x00 | USB (Universal Host Controller Spec) |
| 0x10 | USB (Open Host Controller Spec |
| 0x20 | USB2 Host Controller (Intel Enhanced Host Controller Interface) |
| 0x80 | USB |
| 0xFE | USB (Not Host Controller) |
| 0x04 | 0x00 | Fibre Channel |
| 0x05 | 0x00 | SMBus |
| 0x06 | 0x00 | InfiniBand |
| 0x07 | 0x00 | IPMI SMIC Interface |
| 0x01 | IPMI Kybd Controller Style Interface |
| 0x02 | IPMI Block Transfer Interface |
| 0x08 | 0x00 | SERCOS Interface Standard (IEC 61491) |
| 0x09 | 0x00 | CANbus |
| 0x0D | 0x00 | 0x00 | iRDA Compatible Controller |
| 0x01 | 0x00 | Consumer IR Controller |
| 0x10 | 0x00 | RF Controller |
| 0x11 | 0x00 | Bluetooth Controller |
| 0x12 | 0x00 | Broadband Controller |
| 0x20 | 0x00 | Ethernet Controller (802.11a) |
| 0x21 | 0x00 | Ethernet Controller (802.11b) |
| 0x80 | 0x00 | Other Wireless Controller |
| 0x0E | 0x00 | 0x-- | I20 Architecture |
| 0x00 | Message FIFO |
| 0x0F | 0x01 | 0x00 | TV Controller |
| 0x02 | 0x00 | Audio Controller |
| 0x03 | 0x00 | Voice Controller |
| 0x04 | 0x00 | Data Controller |
| 0x10 | 0x00 | 0x00 | Network and Computing Encrpytion/Decryption |
| 0x10 | 0x00 | Entertainment Encryption/Decryption |
| 0x80 | 0x00 | Other Encryption/Decryption |
| 0x11 | 0x00 | 0x00 | DPIO Modules |
| 0x01 | 0x00 | Performance Counters |
| 0x10 | 0x00 | Communications Syncrhonization Plus Time and Frequency Test/Measurment |
| 0x20 | 0x00 | Management Card |
| 0x80 | 0x00 | Other Data Acquisition/Signal Processing Controller |

Enumerating PCI Buses

There are 3 ways to enumerate devices on PCI buses. The first way is "brute force", checking every device on every PCI bus (regardless of whether the PCI bus exists or not). The second way avoids a lot of work by figuring out valid bus numbers while it scans, and is a little more complex as it involves recursion. For both of these methods you rely on something (firmware) to have configured PCI buses properly (setting up PCI to PCI bridges to forward request from one bus to another). The third method is like the second method, except that you configure PCI bridges while you're doing it.

For all 3 methods, you need to be able to check if a specific device on a specific bus is present and if it is multi-function or not. Pseudo-code might look like this:

void checkDevice(uint8\_t bus, uint8\_t device) {

uint8\_t **function** = 0;

vendorID = getVendorID(bus, device, **function**);

if(vendorID = 0xFFFF) return; *// Device doesn't exist*

checkFunction(bus, device, **function**);

headerType = getHeaderType(bus, device, **function**);

if( (headerType & 0x80) != 0) {

*/\* It is a multi-function device, so check remaining functions \*/*

for(**function** = 1; **function** < 8; **function**++) {

if(getVendorID(bus, device, **function**) != 0xFFFF) {

checkFunction(bus, device, **function**);

}

}

}

}

void checkFunction(uint8\_t bus, uint8\_t device, uint8\_t **function**) {

}

Please note that if you don't check bit 7 of the header type and scan all functions, then some single-function devices will report details for "function 0" for every function.

**"Brute Force" Scan**

For the brute force method, the remaining code is relatively simple. Pseudo-code might look like this:

void checkAllBuses(void) {

uint8\_t bus;

uint8\_t device;

for(bus = 0; bus < 256; bus++) {

for(device = 0; device < 32; device++) {

checkDevice(bus, device);

}

}

}

For this method, there are 32 functions per bus and 256 buses, so you call "checkDevice()" 8192 times.

**Recursive Scan**

The first step for the recursive scan is to implement a function that scans one bus. Pseudo-code might look like this:

void checkBus(uint8\_t bus) {

uint8\_t device;

for(device = 0; device < 32; device++) {

checkDevice(bus, device);

}

}

The next step is to add code in "checkFunction()" that detects if the function is a PCI to PCI bridge. If the device is a PCI to PCI bridge then you want to extract the "secondary bus number" from the bridge's configuration space and call "checkBus()" with the number of the bus on the other side of the bridge.

Pseudo-code might look like this:

void checkFunction(uint8\_t bus, uint8\_t device, uint8\_t **function**) {

uint8\_t baseClass;

uint8\_t subClass;

uint8\_t secondaryBus;

baseClass = getBaseClass(bus, device, **function**);

subClass = getSubClass(bus, device, **function**);

if( (baseClass == 0x06) && (subClass == 0x04) ) {

secondaryBus = getSecondaryBus(bus, device, **function**);

checkBus(secondaryBus);

}

}

The final step is to handle systems with multiple PCI host controllers correctly. Start by checking if the device at bus 0, device 0 is a multi-function device. If it's not a multi-function device, then there is only one PCI host controller and bus 0, device 0, function 0 will be the PCI host controller responsible for bus 0. If it is a multifunction device, then bus 0, device 0, function 0 will be the PCI host controller responsible for bus 0; bus 0, device 0, function 1 will be the PCI host controller responsible for bus 1, etc (up to the number of functions supported).

Pseudo-code might look like this:

void checkAllBuses(void) {

uint8\_t **function**;

uint8\_t bus;

headerType = getHeaderType(0, 0, 0);

if( (headerType & 0x80) == 0) {

*/\* Single PCI host controller \*/*

checkBus(0);

} else {

*/\* Multiple PCI host controllers \*/*

for(**function** = 0; **function** < 8; **function**++) {

if(getVendorID(0, 0, **function**) != 0xFFFF) **break**;

bus = **function**;

checkBus(bus);

}

}

}

**Recursive Scan With Bus Configuration**

This is similar to the recursive scan above; except that you set the "secondary bus" field in PCI to PCI bridges (using something like "setSecondaryBus(bus, device, function, nextBusNumber++);" instead of the "getSecondaryBus();"). However; if you are configuring PCI buses you are also responsible for configuring the memory areas/BARs in PCI functions, and ensuring that PCI bridges forward requests from their primary bus to their secondary buses.

Writing code to support this without a deep understanding of PCI specifications is not recommended; and if you have a deep understanding of PCI specifications you have no need for pseudo code. For this reason there will be no example code for this method here.

IRQ Handling

If you're using the old [PIC](http://wiki.osdev.org/PIC), your life is really easy. You have the *Interrupt Line* field of the header, which is read/write (you can change it's value!) and it says which interrupt will the PCI device fire when it needs attention.

If you plan to use the [I/O APIC](http://wiki.osdev.org/I/O_APIC), your life will be a nightmare. You have 4 new IRQs called INTA#, INTB#, INTC# and INTD#. You can find which IRQ the device will use in the *Interrupt Line* field. In the ACPI AML Tables you will find (using [ACPICA](http://wiki.osdev.org/ACPICA)) that INTA# is connected to a specified interrupt line, INTB# to another, etc...

So far so good. You have, say, 20 devices. 10 of those are using INTA#, 5 for INTB#, 5 for INTC#, and none for INTD#. So when the IRQ number related to #INTC you have to scan the 5 devices to understand who was the interested one. So there is a LOT of IRQ sharing, expecially for INTA#.

With time manufacturers started to use mainly INTA#, forgetting the existence of other pins. So you will likely have 18 devices on INTA# and 2 on INTB#. Motherboard manufacturers decided take the situation in control. So at boot the INTx# are remapped, so that you will have 5 devices for INTA#, 5 for INTB#, 5 for INTC#, and 5 for INTD# (in the best case). That's great! IRQs are balanced and IRQ sharing is reduced. The only problem is that you don't know what devices where mapped. If you read the *Interrupt Pin* you still get INTA#. You know need to parse the MP Tables or the [ACPI](http://wiki.osdev.org/ACPI) ones to solve the mess. Good luck.

Multifunction Devices

Multifunction devices behave in the same manner as normal PCI devices. The easiest way to detect a multifunction device is bit 7 of the header type field - 1 = multifunction. Make sure you mask this bit when you determine header type. To detect the number of functions you need to scan the PCI configuration space for every function - unused functions have vendor 0xFFFF. Device IDs and Class codes vary between functions. Functions are not neccesarily in order - you can have function 0x0, 0x1 and 0x7 in use.

Disclaimer

This text originates from "Pentium on VME", unknown author, md5sum d292807a3c56881c6faba7a1ecfd4c79. The original document is apparently no longer present on the Web ...

Closest match: [[2]](http://www.quicklogic.com/images/appnote10.pdf)

References

* PCI Local Bus Specification, revision 3.0, PCI Special Interest Group, August 12, 2002

See Also

* [PCI Express](http://wiki.osdev.org/PCI_Express)

**External Links**

* <http://lmgtfy.com/?q=pci+local+bus+specification+3.0>
* <http://tldp.org/LDP/tlk/dd/pci.html>
* <http://www.pcidatabase.com/>
* <http://pciids.sourceforge.net/> (More up to date PCI vendor and device numbers)
* <http://www.acm.uiuc.edu/sigops/roll_your_own/7.c.html>
* <http://tldp.org/LDP/tlk/dd/pci.html>
* <http://msdn.microsoft.com/en-us/library/ms903537.aspx>
* <http://www.pcisig.com/specifications/conventional/ECN_SATA_Class_Code.pdf>

[Category](http://wiki.osdev.org/Special:Categories):

* [Buses](http://wiki.osdev.org/Category:Buses)
* [Log in](http://wiki.osdev.org/index.php?title=Special:UserLogin&returnto=PCI)
* [Page](http://wiki.osdev.org/PCI)
* [Discussion](http://wiki.osdev.org/Talk:PCI)
* [Read](http://wiki.osdev.org/PCI)
* [View source](http://wiki.osdev.org/index.php?title=PCI&action=edit)
* [View history](http://wiki.osdev.org/index.php?title=PCI&action=history)

Top of Form



Bottom of Form

Navigation

* [Main Page](http://wiki.osdev.org/Main_Page)
* [Forums](http://forum.osdev.org/)
* [FAQ](http://wiki.osdev.org/Category:FAQ)
* [OS Projects](http://wiki.osdev.org/Projects)
* [Random page](http://wiki.osdev.org/Special:Random)

About

* [This site](http://wiki.osdev.org/OSDevWiki:About)
* [Joining](http://wiki.osdev.org/OSDevWiki:Joining)
* [Editing help](http://wiki.osdev.org/OSDevWiki:Editing)
* [Recent changes](http://wiki.osdev.org/Special:RecentChanges)

Toolbox

* [What links here](http://wiki.osdev.org/Special:WhatLinksHere/PCI)
* [Related changes](http://wiki.osdev.org/Special:RecentChangesLinked/PCI)
* [Special pages](http://wiki.osdev.org/Special:SpecialPages)
* [Printable version](http://wiki.osdev.org/index.php?title=PCI&printable=yes)
* [Permanent link](http://wiki.osdev.org/index.php?title=PCI&oldid=15355)

In other languages

* [Deutsch](http://www.lowlevel.eu/wiki/Peripheral_Component_Interconnect)
* This page was last modified on 25 October 2013, at 13:13.
* This page has been accessed 116,225 times.
* [Privacy policy](http://wiki.osdev.org/OSDev_Wiki:Privacy_policy)
* [About OSDev Wiki](http://wiki.osdev.org/OSDev_Wiki:About)
* [Disclaimers](http://wiki.osdev.org/OSDev_Wiki:General_disclaimer)