PCI

Contents

The PCI Bus

The PCI (Peripheral Component Interconnect (https://en.wikipedia.org/wiki/Conventional_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 signalling environments, the PCI bus meets the needs of both low end desktop systems and 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 backwards compatibility. Only configuration mechanism #1 will be described here, as it is the only access mechanism that will be used in the future.

Configuration Space Access 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).

The least significant byte selects the offset into the 256-byte configuration space available through this method. Since all reads and writes must be both 32-bits and aligned to work on all implementations, the two lowest bits of CONFIG_ADDRESS must always be zero, with the remaining six bits allowing you to choose each of the 64 32-bit words. If you don't need all 32 bits, you'll have to perform the unaligned access in software by aligning the address, followed by masking and shifting the answer.

Bit 31	Bits 30-24	Bits 23-16	Bits 15-11	Bits 10-8	Bits 7-0
Enable Bit	Reserved	Bus Number	Device Number	Function Number	Register Offset[1]

[1] Register Offset has to point to consecutive DWORDs, ie. bits 1:0 are always 0b00 (they are still part of the Register Offset).

The following code segment illustrates the use of configuration mechanism #1 to read 16-bit fields from configuration space. Note that this segment, the outl(port, value) and inl(port) functions refer to the OUTL and INL Pentium assembly language instructions.

```
uint16_t pciConfigReadWord(uint8_t bus, uint8_t slot, uint8_t func, uint8_t offset) {
    uint32_t address;
    uint32_t lbus  = (uint32_t)bus;
    uint32_t lslot = (uint32_t)slot;
    uint32_t lfunc = (uint32_t)func;
    uint16_t tmp = 0;

    // Create configuration address as per Figure 1
    address = (uint32_t)((lbus << 16) | (lslot << 11) |
              (lfunc << 8) | (offset & 0xFC) | ((uint32_t)0x80000000));

    // Write out the address
    outl(0xCF8, address);
    // Read in the data
    // (offset & 2) * 8) = 0 will choose the first word of the 32-bit register
    tmp = (uint16_t)((inl(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.

```
uint16_t pciCheckVendor(uint8_t bus, uint8_t slot) {
    uint16_t 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);
}
```

Configuration Space Access Mechanism #2

This configuration space access mechanism was deprecated in PCI version 2.0. This means it's only likely to exist on hardware from around 1992 (when PCI 1.0 was introduced) to 1993 (when PCI 2.0 was introduced), which limits it to 80486 and early Pentium motherboards.

For access mechanism #2, the IO port at `0xCF8` is an 8-bit port and is used to enable/disable the access mechanism and set the function number. It has the following format:

Bits 7-4	Bits 3-1	Bit 0

Key (0 = access mechanism disabled, non-zero = access mechanism enabled)	Function number	Special cycle enabled if set

The IO port at `0xCFA` (the "Forwarding Register") is also an 8-bit port, and is used to set the bus number for subsequent PCI configuration space accesses.

Once the access mechanism has been enabled; accesses to IO ports `0xC000` to `0xCFFF` are used to access PCI configuration space. The IO port number has the following format:

Bits 15-12	Bits 11-8	Bits 7-2	Bits 1-0
Must be 1100b	Device number	Register index	Must be zero

Note that this limits the system to 16 devices per PCI bus.

Memory Mapped PCI Configuration Space Access

PCI Express introduced a new way to access PCI configuration space, where it's simply memory mapped and no IO ports are used. This access mechanism is described in PCI Express.

Note that systems that do provide the memory mapped access mechanism are also required to support PCI access mechanism #1 for backwards compatibility.

Detecting Configuration Space Access Mechanism/s

In general there are 4 cases:

- Computer doesn't support PCI (either the computer is too old, or your OS is being run at some time in the future after PCI has been superseded)
- Computer supports mechanism #2
- Computer supports mechanism #1 but doesn't support the memory mapped access mechanism
- Computer supports both mechanism #1 and the memory mapped access mechanism

For BIOS systems, `int 0x1A, AX=0xB101` will tell you if the system uses mechanism #1 or mechanism #2. If this function doesn't exist you can't be sure if the computer supports PCI or not. If it says mechanism #1 is supported you won't know if the memory mapped access mechanism is also supported or not.

For UEFI systems, it's extremely safe to assume that mechanism #2 is not supported; and you can test to see if the computer supports PCI or not by checking to see if the "PCI bus support" protocol exists. If PCI is supported, there's no easy way to determine if (e.g.) the computer supports mechanism #1 or not.

For both BIOS and UEFI systems, you can check the ACPI tables to determine if the memory mapped access mechanism is supported.

This leaves a few cases uncovered (e.g. where you don't know if whether mechanism #1 or #2 are supported despite trying all of the above). For these cases the only option left is manual probing. This means 2 specific tests - whether mechanism #1 is supported, and if not whether mechanism #2 is supported. Please note that manual probing has risks; in that if there is no PCI (e.g. the system only has ISA) the IO port accesses might cause undefined behaviour (especially on systems where the ISA bus ignores highest 6 bits of the IO port address, where accessing IO port `0xCF8` is the same as accessing IO port `0xF8`).

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.

Common Header Fields

The following field descriptions are common to all Header Types:

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
...					

- *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 (the list is here (https://pcisig.com/membership/member-companies)) 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(Programming Interface Byte):* 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 beginning at byte `0x10` of the header. If bit 7 of this register is set, the device has multiple functions; otherwise, it is a single function device. Types:
  - `0x0`: a general device
  - `0x1`: a PCI-to-PCI bridge
  - `0x2`: a PCI-to-CardBus bridge.
- *Latency Timer:* Specifies the latency timer in units of PCI bus clocks.
- *Cache Line Size:* Specifies the system cache line size in 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.

Remember 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.

Command Register

Here is the layout of the Command register:

Bits 11-15	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Interrupt 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
	RW	RO	RW	RO	RW	RO	RO	RO	RW	RW	RW

- *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.

If the kernel configures the BARs of the devices, the kernel also have to enable bits 0 and 1 for it to activate.

Status Register

Here is the layout of the Status register:

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bits 9-10	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bits 0-2
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

RW1C	RW1C	RW1C	RW1C	RW1C	RO	RW1C	RO	RO	RO	RO	RO	

- *Detected Parity Error* - This bit will be set to 1 whenever the device detects a parity error, even if parity error handling is disabled.
- *Signalled 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.
- *Signalled 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 `0x0` represents fast timing, a value of `0x1` represents medium timing, and a value of `0x2` 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.

Header Type 0x0

This table is applicable if the Header Type is `0x0`. (Figure 2)

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
0x4	0x10	Base address #0 (BAR0)			
0x5	0x14	Base address #1 (BAR1)			
0x6	0x18	Base address #2 (BAR2)			
0x7	0x1C	Base address #3 (BAR3)			
0x8	0x20	Base address #4 (BAR4)			
0x9	0x24	Base address #5 (BAR5)			
0xA	0x28	Cardbus CIS Pointer			
0xB	0x2C	Subsystem ID		Subsystem Vendor ID	
0xC	0x30	Expansion ROM base address			
0xD	0x34	Reserved			Capabilities Pointer
0xE	0x38	Reserved			
0xF	0x3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line

The following field descriptions apply if the Header Type is `0x0`:

- *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 `0x1` is INTA#, `0x2` is INTB#, `0x3` is INTC#, `0x4` is INTD#, and `0x0` 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 (i.e. an offset into this function's configuration space) 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.

Header Type 0x1 (PCI-to-PCI bridge)

This table is applicable if the Header Type is `0x1` (PCI-to-PCI bridge) (Figure 3)

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
0x4	0x10	Base address #0 (BAR0)			
0x5	0x14	Base address #1 (BAR1)			
0x6	0x18	Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number
0x7	0x1C	Secondary Status		I/O Limit	I/O Base
0x8	0x20	Memory Limit		Memory Base	
0x9	0x24	Prefetchable Memory Limit		Prefetchable Memory Base	
0xA	0x28	Prefetchable Base Upper 32 Bits			
0xB	0x2C	Prefetchable Limit Upper 32 Bits			
0xC	0x30	I/O Limit Upper 16 Bits		I/O Base Upper 16 Bits	
0xD	0x34	Reserved			Capability Pointer
0xE	0x38	Expansion ROM base address			
0xF	0x3C	Bridge Control		Interrupt PIN	Interrupt Line

Header Type Register

Here is the layout of the Header Type register:

Bit 7	Bits 6-0
MF	Header Type

- *MF* - If MF = 1 Then this device has multiple functions.
- *Header Type* - `0x0` Standard Header - `0x1` PCI-to-PCI Bridge - `0x2` CardBus Bridge

BIST Register

Here is the layout of the BIST register:

Bit 7	Bit 6	Bits 4-5	Bits 0-3
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.

Header Type 0x2 (PCI-to-CardBus bridge)

This table is applicable if the Header Type is `0x2` (PCI-to-CardBus bridge)

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
0x4	0x10	CardBus Socket/ExCa base address			
0x5	0x14	Secondary status		Reserved	Offset of capabilities list

0x6	0x18	CardBus latency timer	Subordinate bus number	CardBus bus number	PCI bus number
0x7	0x1C	Memory Base Address 0			
0x8	0x20	Memory Limit 0			
0x9	0x24	Memory Base Address 1			
0xA	0x28	Memory Limit 1			
0xB	0x2C	I/O Base Address 0			
0xC	0x30	I/O Limit 0			
0xD	0x34	I/O Base Address 1			
0xE	0x38	I/O Limit 1			
0xF	0x3C	Bridge Control		Interrupt PIN	Interrupt Line
0x10	0x40	Subsystem Vendor ID		Subsystem Device ID	
0x11	0x44	16-bit PC Card legacy mode base address			

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

Bits 31-4	Bit 3	Bits 2-1	Bit 0
16-Byte Aligned Base Address	Prefetchable	Type	Always 0

I/O Space BAR Layout

Bits 31-2	Bit 1	Bit 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 `0x0` then the base register is 32-bits wide and can be mapped anywhere in the 32-bit Memory Space. A value of `0x2` 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 `0x1` 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 a base address register is marked as Prefetchable, it means that the region does not have read side effects (reading from that memory range doesn't change any state), and it is allowed for the CPU to cache loads from that memory region and read it in bursts (typically cache line sized). Hardware is also allowed to merge repeated stores to the same address into one store of the latest value. If you are using paging and want maximum performance, you should map prefetchable MMIO regions as WT (write-through) instead of UC (uncacheable). On x86, frame buffers are the exception, they should be almost always be mapped WC (write-combining).

Address and size of the BAR

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)`.

Before attempting to read the information about the BAR, make sure to disable both I/O and memory decode in the command byte. You can restore the original value after completing the BAR info read. This is needed as some devices are known to decode the write of all ones to the register as an (unintended) access.

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 (0x1000000 after decoding) and you can only modify the upper 8-bits. [1] (https://web.archive.org/web/20150101180929/https://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 details most of the known device types and functions:

Class Code	Subclass	Prog IF
0x0 - Unclassified	0x0 - Non-VGA-Compatible Unclassified Device	--
	0x1 - VGA-Compatible Unclassified Device	--
0x1 - Mass Storage Controller	0x0 - SCSI Bus Controller	--
	0x1 - IDE Controller	0x0 - ISA Compatibility mode-only controller
		0x5 - PCI native mode-only controller
		0xA - ISA Compatibility mode controller, supports both channels switched to PCI native mode
		0xF - PCI native mode controller, supports both channels switched to ISA compatibility mode
		0x80 - ISA Compatibility mode-only controller, supports bus mastering
		0x85 - PCI native mode-only controller, supports bus mastering
		0x8A - ISA Compatibility mode controller, supports both channels switched to PCI native mode, supports bus mastering
		0x8F - PCI native mode controller, supports both channels switched to ISA compatibility mode, supports bus mastering
	0x2 - Floppy Disk Controller	--
	0x3 - IPI Bus Controller	--
	0x4 - RAID Controller	--
	0x5 - ATA Controller	0x20 - Single DMA
		0x30 - Chained DMA
	0x6 - Serial ATA Controller	0x0 - Vendor Specific Interface
		0x1 - AHCI 1.0
		0x2 - Serial Storage Bus
	0x7 - Serial Attached SCSI Controller	0x0 - SAS
		0x1 - Serial Storage Bus
	0x8 - Non-Volatile Memory Controller	0x1 - NVMHCI
		0x2 - NVM Express
	0x80 - Other	--
0x2 - Network Controller	0x0 - Ethernet Controller	--
	0x1 - Token Ring Controller	--
	0x2 - FDDI Controller	--
	0x3 - ATM Controller	--
	0x4 - ISDN Controller	--
	0x5 - WorldFip Controller	--
	0x6 - PICMG 2.14 Multi Computing Controller	--
	0x7 - Infiniband Controller	--
	0x8 - Fabric Controller	--
	0x80 - Other	--
0x3 - Display Controller	0x0 - VGA Compatible Controller	0x0 - VGA Controller
		0x1 - 8514-Compatible Controller
	0x1 - XGA Controller	--
	0x2 - 3D Controller (Not VGA-Compatible)	--
	0x80 - Other	--
0x4 - Multimedia Controller	0x0 - Multimedia Video Controller	--
	0x1 - Multimedia Audio Controller	--
	0x2 - Computer Telephony Device	--
	0x3 - Audio Device	--

	0x80 - Other	--
0x5 - Memory Controller	0x0 - RAM Controller	--
	0x1 - Flash Controller	--
	0x80 - Other	--
0x6 - Bridge	0x0 - Host Bridge	--
	0x1 - ISA Bridge	--
	0x2 - EISA Bridge	--
	0x3 - MCA Bridge	--
	0x4 - PCI-to-PCI Bridge	0x0 - Normal Decode
		0x1 - Subtractive Decode
	0x5 - PCMCIA Bridge	--
	0x6 - NuBus Bridge	--
	0x7 - CardBus Bridge	--
	0x8 - RACEway Bridge	0x0 - Transparent Mode
		0x1 - Endpoint Mode
	0x9 - PCI-to-PCI Bridge	0x40 - Semi-Transparent, Primary bus towards host CPU
		0x80 - Semi-Transparent, Secondary bus towards host CPU
	0x0A - InfiniBand-to-PCI Host Bridge	--
	0x80 - Other	--
0x7 - Simple Communication Controller	0x0 - Serial Controller	0x0 - 8250-Compatible (Generic XT)
		0x1 - 16450-Compatible
		0x2 - 16550-Compatible
		0x3 - 16650-Compatible
		0x4 - 16750-Compatible
		0x5 - 16850-Compatible
		0x6 - 16950-Compatible
	0x1 - Parallel Controller	0x0 - Standard Parallel Port
		0x1 - Bi-Directional Parallel Port
		0x2 - ECP 1.X Compliant Parallel Port
		0x3 - IEEE 1284 Controller
		0xFE - IEEE 1284 Target Device
	0x2 - Multiport Serial Controller	--
	0x3 - Modem	0x0 - Generic Modem
		0x1 - Hayes 16450-Compatible Interface
		0x2 - Hayes 16550-Compatible Interface
		0x3 - Hayes 16650-Compatible Interface
		0x4 - Hayes 16750-Compatible Interface
	0x4 - IEEE 488.1/2 (GPIB) Controller	--
	0x5 - Smart Card Controller	--
	0x80 - Other	--
0x8 - Base System Peripheral	0x0 - PIC	0x0 - Generic 8259-Compatible
		0x1 - ISA-Compatible
		0x2 - EISA-Compatible
		0x10 - I/O APIC Interrupt Controller
		0x20 - I/O(x) APIC Interrupt Controller
	0x01 - DMA Controller	0x00 - Generic 8237-Compatible
		0x01 - ISA-Compatible
		0x02 - EISA-Compatible
	0x02 - Timer	0x00 - Generic 8254-Compatible
		0x01 - ISA-Compatible

		0x02 - EISA-Compatible
		0x03 - HPET
	0x3 - RTC Controller	0x0 - Generic RTC
		0x1 - ISA-Compatible
	0x4 - PCI Hot-Plug Controller	--
	0x5 - SD Host controller	--
	0x6 - IOMMU	--
	0x80 - Other	--
0x9 - Input Device Controller	0x0 - Keyboard Controller	--
	0x1 - Digitizer Pen	--
	0x2 - Mouse Controller	--
	0x3 - Scanner Controller	--
	0x4 - Gameport Controller	0x0 - Generic
		0x10 - Extended
	0x80 - Other	--
0xA - Docking Station	0x0 - Generic	--
	0x80 - Other	--
0xB - Processor	0x0 - 386	--
	0x1 - 486	--
	0x2 - Pentium	--
	0x3 - Pentium Pro	--
	0x10 - Alpha	--
	0x20 - PowerPC	--
	0x30 - MIPS	--
	0x40 - Co-Processor	--
	0x80 - Other	--
0xC - Serial Bus Controller	0x0 - FireWire (IEEE 1394) Controller	0x0 - Generic
		0x10 - OHCI
	0x1 - ACCESS Bus Controller	--
	0x2 - SSA	--
	0x3 - USB Controller	0x0 - UHCI Controller
		0x10 - OHCI Controller
		0x20 - EHCI (USB2) Controller
		0x30 - XHCI (USB3) Controller
		0x80 - Unspecified
		0xFE - USB Device (Not a host controller)
	0x4 - Fibre Channel	--
	0x5 - SMBus Controller	--
	0x6 - InfiniBand Controller	--
	0x7 - IPMI Interface	0x0 - SMIC
		0x1 - Keyboard Controller Style
		0x2 - Block Transfer
	0x8 - SERCOS Interface (IEC 61491)	--
	0x9 - CANbus Controller	--
	0x80 - Other	--
0xD - Wireless Controller	0x0 - iRDA Compatible Controller	--
	0x1 - Consumer IR Controller	--
	0x10 - RF Controller	--
	0x11 - Bluetooth Controller	--
	0x12 - Broadband Controller	--
	0x20 - Ethernet Controller (802.1a)	--

	0x21 - Ethernet Controller (802.1b)	--
	0x80 - Other	--
0xE - Intelligent Controller	0x0 - I20	--
0xF - Satellite Communication Controller	0x1 - Satellite TV Controller	--
	0x2 - Satellite Audio Controller	--
	0x3 - Satellite Voice Controller	--
	0x4 - Satellite Data Controller	--
0x10 - Encryption Controller	0x0 - Network and Computing Encrpytion/Decryption	--
	0x10 - Entertainment Encryption/Decryption	--
	0x80 - Other	--
0x11 - Signal Processing Controller	0x0 - DPIO Modules	--
	0x1 - Performance Counters	--
	0x10 - Communication Synchronizer	--
	0x20 - Signal Processing Management	--
	0x80 - Other	--
0x12 - Processing Accelerator	--	--
0x13 - Non-Essential Instrumentation	--	--
0x14 - 0x3F (Reserved)	--	--
0x40 - Co-Processor	--	--
0x41 - 0xFE (Reserved)	--	--
0xFF - Unassigned Class (Vendor specific)	--	--

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's 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) {
    uint16_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 devices 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 == 0x6) && (subClass == 0x4)) {
        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's a multi-function 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.

Configuring PCI-to-PCI bridges

To configure this the kernel has to forget about BIOS for a moment, first scan the root PCI device, (check if it is multi-function to scan multiple buses). Root bus is always 0.

Secondary and subordinate bus acts as a range start-end of what buses the PCI-to-PCI bridge will manage.

Then, after this step it's up to implementation: Scan each device, then if a bridge is found, allocate a bus number to it (note: PCI-to-PCI bridges can have multiple bridges within them). Scan that bus and find more devices, once you find more bridges add 1 to the subordinate bus for each bridge found, because PCI-to-PCI bridges can manage multiple bridges.

And this is just the beginning: After allocating bus numbers, you need to allocate MMIO, it would be trivial if it wasn't for the fact that PCI has 3 areas the kernel manages: IO, Prefetch and Memory.

A bridge can manage multiple buses, but that means it spans all the memory of these buses, if device 1 is behind bridge 2, which is behind bridge 1, then bridge 2 will contain the memory area of device 1 + any other device's areas, supposing IO is 4M, Memory is 16M and Prefetch is 5MB (supposing there are 3 devices in bridge's 2 bus), bridge 2 would contain those, take in reference table for Header type `0x1`. However, bridge 1 will contain the areas of bridge 2 + any other devices in bridge's 1 bus.

Once all memory areas are allocated, the devices can be used. Note that PCI-to-PCI bridges also have BAR's.

If the kernel does not configure a PCI-to-PCI bridge, the BIOS will probably do, however on environments without BIOS, this method is mandatory otherwise devices behind that bridge won't show up.

IRQ Handling

If you're using the old 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, things aren't so easy. Basically the PCI bus specifies that there are 4 interrupt pins. They are labeled INTA#, INTB#, INTC#, and INTD#. You find out what pin a device is using by reading the *Interrupt Pin* field of the header. So far, so good.

The only problem is that the PCI pins correspond to an arbitrary I/O APIC pin. It's up to the programmer to find the mapping. How is that done? You must parse the MP Tables or the ACPI tables. The MP tables are easy, only they aren't supported on newer hardware. The ACPI tables, however, involve parsing AML, which is not an easy task. If one wants to take a shortcut, you can use ACPICA.

Once you've found the I/O APIC pin, all you do is map that to an IRQ using the I/O APIC redirection table. See the I/O APIC article for more information on this.

Alternatively, you could just use MSI or MSI-X, and skip complicated ACPI.

Message Signaled Interrupts

Message Signaled Interrupts, or MSI, have been supported since PCI 2.2. However, support for them is *mandatory* in PCIe devices, so you can be sure that they're usable on modern hardware. There are two versions of MSI implementation. First implementation called MSI and its evolution called MSI-X. They have different PCI capability and different ways how to handle multiple device interrupts and masking of such interrupts. The both implementations are exclusive and either MSI or MSI-X needs to be enabled.

Unlike legacy PCI interrupt, which is level triggered and routed usually through I/O APIC pin, the device might support multiple interrupt sources with MSI or MSI-X or both. The PCI device signals the interrupt by issuing a 32-bit write transaction on PCI bus with the target address taken from address data pair described below.

The legacy MSI supports single address data pair (Message Address / Message Data) with some editing of data allowed if multiple device interrupts are enabled. The MSI-X has for each device interrupt own address/data pair.

The message address/data pair needs to be supplied by the driver/OS and needs to target some special hardware register which in turn generates the interrupt. Therefore the exact values of address and data pairs are specific to the architecture. The format is described in detail in the MSI Register Format.

Enabling MSI

First, check that the device has a pointer to the capabilities list (status register bit 4 set to 1). Then, traverse the capabilities list. The low 8 bits of a capability register are the ID - `0x05` for MSI. The next 8 bits are the offset (in PCI Configuration Space) of the next capability.

The MSI capability is as follows:

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Cap + 0x0	Cap + 0x0	Message Control		Next pointer	Capability ID = 05
Cap + 0x1	Cap + 0x4	Message Address [Low]			
Cap + 0x2	Cap + 0x8	[Message Address High]			

Cap + 0x2/0x3	Cap + 0x8/0xC	Reserved	Message Data
Cap + 0x4	Cap + 0x10	[Mask]	
Cap + 0x5	Cap + 0x14	[Pending]	

Here is the layout of the message control register:

Bits 15-9	Bit 8	Bit 7	Bits 6-4	Bits 3-1	Bit 0
Reserved	Per-vector masking	64-bit	Multiple Message Enable	Multiple Message Capable	Enable

Multiple messages:

MME / MMI	Interrupts
000	1
001	2
010	4
011	8
100	16
101	32

In MME, specifies the number of low bits of Message Data that may be modified by the device.

Therefore, the interrupt vector block allocated must be aligned accordingly.

Interrupt masking

If capable, you can mask individual messages by setting the corresponding bit (1 << n), in the mask register.

If a message is pending, then the corresponding bit in the pending register is set.

Note that the PCI specification doesn't specify the location of these registers if the message address is 32-bit. This is because a function that supports masking is required to implement 64-bit addressing!

Enabling MSI-X

Like for MSI, you have to find the MSI-X capability, but the ID for MSI-X is **0x11**

The structure is as follows:

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-3	Bits 2-0
Cap + 0x0	Cap + 0x0	Message Control		Next Pointer	Capability ID = 11	
Cap + 0x1	Cap + 0x4	Table Offset				BIR
Cap + 0x2	Cap + 0x8	Pending Bit Offset				Pending Bit BIR

Unlike MSI, MSI-X supports 2048 interrupts. This is achieved by maintaining a table of interrupts in the PCI device's address space. The wording of the PCI 3.0 specification indicates that this *must* be via a Memory BAR.

BIR specifies which BAR is used for the Message Table. This may be a 64-bit BAR, and is zero-indexed (so BIR=0, BAR0, offset `0x10` into the header).

Table Offset is an offset into that BAR where the Message Table lives. Note that it is 8-byte aligned - so simply mask BIR.

The format of Message Control is as follows:

Bit 15	Bit 14	Bits 13-11	Bits 10-0
Enable	Function Mask	Reserved	Table Size

Table Size is N - 1 encoded, and is the number of entries in the MSI-X table. This field is Read-Only.

Now you have all the information you need to find the MSI-X table:

Bits 127-96	Bits 95-64	Bits 63-32	Bits 31-0
Vector Control (0)	Message Data (0)	Message Address High (0)	Message Address Low (0)
Vector Control (1)	Message Data (1)	Message Address High (1)	Message Address Low (1)
...	...	...	...
Vector Control (N - 1)	Message Data (N - 1)	Message Address High (N - 1)	Message Address Low (N - 1)

Vector Control is as follows:

Bits 31-1	Bit 0
Reserved	Masked

Note that **Message Address** is is 4-byte aligned, so, again, mask the low bits. The interrupt is masked if **Masked** is set to 1.

Message Address and Data are as they were for MSI - architecture specific. However, unlike with MSI, you can specify independent vectors for all the interrupts, only limited by having the same upper 32-bit message address.

Multi-function Devices

Multi-function devices behave in the same manner as normal PCI devices. The easiest way to detect a multi-function device is bit 7 of the header type field. If it is set (value = `0x80`), the device is multi-function -- else it is not. 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 necessarily 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] (https://web.archive.org/web/20071009221818/http://www.quicklogic.com/images/appnote10.pdf)

References

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

See Also

Articles

- PCI Express

External Links

- Very useful page about VendorID and DeviceID (https://devicehunt.com/about)
- Very detailed PCI-to-PCI architecture specification (https://cds.cern.ch/record/551427/files/cer-2308933.pdf)
- How PCI Express devices talk (http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1)
- PCI Local Bus (Solaris documentation) (https://docs.oracle.com/cd/E19120-01/open.solaris/819-3196/hwovr-22/index.html)
- PCI in the Linux kernel (https://tldp.org/LDP/tlk/dd/pci.html)
- More up to date PCI vendor and device numbers (https://pci-ids.ucw.cz)
- Structure describes that PCI configuration space for PCI devices (https://msdn.microsoft.com/en-us/library/ms903537.aspx)