Contents

[1. PCI Configuration Space: - 2](#_Toc126531094)

[1.1. Configuration header: - 3](#_Toc126531095)

[1.2. 192 bytes (40h – FFh):- 5](#_Toc126531096)

[2. Permits Plug-And-Play: - 5](#_Toc126531097)

[3. Types of configuration space: - 5](#_Toc126531098)

[4. Access to Configuration Space: - 6](#_Toc126531099)

[4.1. Configuration of PCI compatible configuration space: - 6](#_Toc126531100)

[4.2. I/O Port CONFIG\_ADDRESS (CF8h): - 6](#_Toc126531101)

[4.3. I/O Port CONFIG\_DATA (CFCh): - 7](#_Toc126531102)

[4.4. PCI Device Identification: - 7](#_Toc126531103)

[5. Configuration Space register details: - 8](#_Toc126531104)

[5.1. Command register: - 9](#_Toc126531105)

[5.2. Status Register: - 10](#_Toc126531106)

[5.3. Class Code register: - 11](#_Toc126531107)

[5.4. Header Type register: - 12](#_Toc126531108)

[5.4.1 Header Type 0x0 13](#_Toc126531109)

[5.4.2 Header Type 0x1 (PCI-to-PCI bridge) 14](#_Toc126531110)

[5.4.3 Header Type 0x2 (PCI-to-CardBus bridge) 15](#_Toc126531111)

[5.5. BIST register: - 16](#_Toc126531112)

[5.6. Interrupt Line register: - 16](#_Toc126531113)

[5.7. Capability Pointer Register: - 16](#_Toc126531114)

[5.8. BAR Registers: - 17](#_Toc126531115)

[5.8.1. Address and size of BAR: - 19](#_Toc126531116)

[6. Accessing configuration space: - 23](#_Toc126531117)

[6.1. Configuration space access in legacy PCI device 23](#_Toc126531118)

[6.2. Configuration space access in PCIe device 26](#_Toc126531119)

[6.2.1. Enhanced Configuration Mechanism 26](#_Toc126531120)

# PCI Configuration Space: -

According to PCI specification, PCI devices or target should undergo software driven device initialization and configuration. So,  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.

PCI devices have a set of registers referred to as configuration space. These registers are then mapped to memory locations such as the I/O Address Space of the CPU. On PCI Express buses, this configuration space may be referred to as the the Extended Configuration Space. Every PCI device in the system, including the PCI-PCI bridges has a configuration space.

Size of configuration space in PCI – 256 bytes

Size of configuration space in PCIe – 4096 bytes

In PCIe, out of 4k bytes, the first 256 bytes of configuration space will have general information of the device, remaining bytes are used for class specific information. This initial 256 bytes is same as that of PCI for backward compatibility. Every PCIe device has its configuration space mapped to memory.

In PCI, configuration space of 256 bytes is addressed by using 8-bit PCI bus, 5-bit device and 3-bit function numbers for the device, it is commonly known as bus/device/funtion or BDF.

Again these 256 bytes have two sub-categories: -

1. First 64 bytes (00h – 3Fh) is called Standard Configuration Header. I had PCI ID i.e., Vendor ID and Product ID registers, to identify device.
2. Remaining 192 byes ( 40h – FFh) represent user-definable configuration space, such as information specific to a PC card. I also have capabilities supported by device.

PCI Express (PCIe) has extended configuration space up to 4096 bytes. The only standardized part of extended configuration space is the first four bytes at 0x100 which are the start of an extended capability list. Extended capabilities are very much like normal capabilities except that they can refer to any byte in the extended configuration space (by using 12 bits instead of eight). Extended capabilities have a four-bit version number and a 16-bit capability ID. Extended capability IDs overlap with normal capability IDs, but there is no chance of confusion as they are in separate lists.

Diagram

Description automatically generated

Figure – 1

## Configuration header: -

Out of 256 bytes configuration space, first 64 bytes (00h – 3Fh) of configuration space are standardized, also called as “Configuration Header”.

In figure – 1, in 64 bytes configuration header, the black ones are read only and white ones are writable registers where the BSP code writes down the address details, configure this device and map it to a particular address space. Address ranges are assigned at boot time.

We can in 64 bytes configuration header, there resides PCI ID i.e., Vendor ID and Device ID registers, to identify the device.

If an Endpoint has 7 functionalities for example, then there will be seven 256 bytes configuration space regions.

Table

Description automatically generated

Figure – 2: PCI Configuration Header

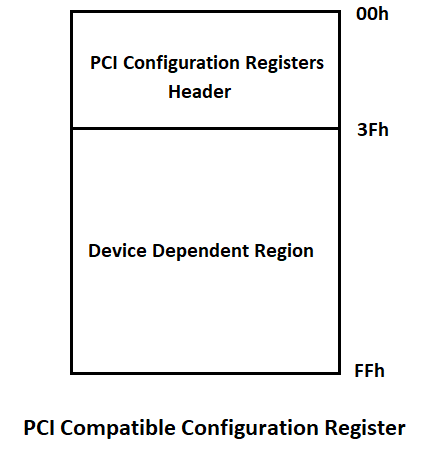


Figure - 3

## 192 bytes (40h – FFh):-

The remaining 192 bytes (40h – FFh) out of 256 bytes represent user-definable configuration space, such as the information specific to PC card for use by its accompanying software. These 192 bytes defines capabilities using various registers.

It consists of device specific information as per PCI-SIG documentation.

# Permits Plug-And-Play: -

1. Base address registers allow an agent to be mapped dynamically into memory or I/O space.
2. A programmable interrupt-line setting allows a software driver to program a PC card with an IRQ upon power up.

# Types of configuration space: -

There are two types of configurations spaces: -

1. Type 0 Configuration Space – for Endpoints
2. Type 1 Configuration Space – for Root Complexes and Switches

# Access to Configuration Space: -

There are two ways to access PCI comaptible 256 bytes configuration space registers:

* Port I/O or Memory-mapped I/O

But bytes in configuration space other than first 256 bytes i.e., 257th byte to 4096tH byte can only be accessed using “Memory-mapped I/O”.

Access to Configuration Space will be done via Port I/O in below situations: -

1. When in Real Mode access to 32-bit memory space is limited.
2. If PCIEXBAR register is not configured, then access happens in Port I/O, as PCIEXBAR register enables MMIO.

## Configuration of PCI compatible configuration space: -

PCI configuration transaction is generated by CPU/BIOS for 256 bytes PCI compatible registers. CPU/BIOS program configures the registers to set device and system parameters.

Compatible PCI is configured using PORT I/O Address/Data pair (CONFIG\_ADDRESS, CONFIG\_DATA)

Two 32 bit I/O locations are used to generate configuration transactions: -

* CF8h (CONFIG\_ADDRESS)
* CFCh (CONFIG\_DATA)

They are mentioned as CF8/CFC in Intel chipset datasheets.

## I/O Port CONFIG\_ADDRESS (CF8h): -

Table

Description automatically generated

Figure - 4

* If Bit 31 is set to 1, then all read and writes to CONFIG\_DATA are PCI Configuration transaction.
* Bits 30:24 are read only and must return 0.
* Bits 23:16 select specific bus (upto 256 buses)
* Bits 15:11 select a device in the specific bus (upto 32 devices)
* Bits 10:8 select specific function of a device (upto 8 devices)
* Bits 7:0 select an offset within the configuration space (256 bytes max), DWORD aligned as bits 1:0 are hard coded to 0.
* Addresses are given in B/D/F format, Offset. It is also written as B:D:F, Offset
* We should visualize PCI configuration header , when we talk about access to specific register number / offset in CONFIG\_ADDRESS

## I/O Port CONFIG\_DATA (CFCh): -

* CONFIG\_DATA can be accessed in DWORD(32 bits), WORD, or BYTE configurations.
* Reads and Writes to CONFIG\_DATA with Bit 31 in CONFIG\_ADDRESS set/enabled results in PCI transaction to the device specified in CONFIG\_ADDRESS.
* PCI spec says that if Bit 31 is not enabled, then the transaction is forwarded out as Port I/O.

## PCI Device Identification: -

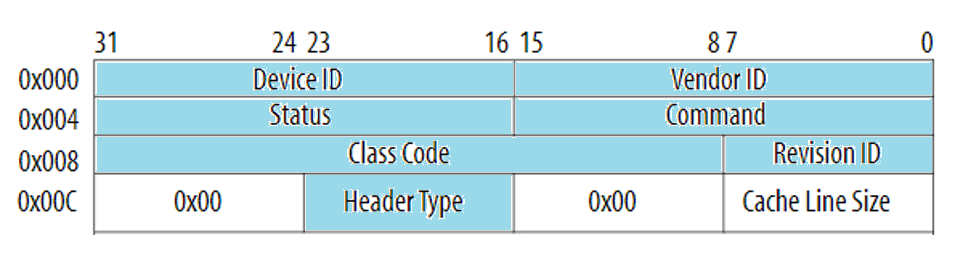


Figure – 5

Below five fields shown in figure – 5 highlighted in sky colour can be used to identify the device and its basic fucntionalities.

1. Vendor ID : - It is used to identify manufacturer of device. Vendor ID is allocated by PCI-SIG.
2. Device ID: - Device ID identifies the device, set by the vendor.
3. Revision ID: - Revision ID is set by the vendor, it is viewed as extension to Device ID.

Ex. – AMD controller is 1022h.

1. Class code: - Class code is used to identify the generic functionality of device.
2. Header type: - Header type identifies what type of header to expect( ex – general, PCI bridge, Cardbus bridge). If Bit – 7 is set (0x80), then it indicates device is a multi-function device.

Header type: 0x0 means Standard header

0x1 means PCI-to-PCI bridge

0x2 means CardBus bridge

# Configuration Space register details: -

Both type 0 and type 1 configurations have a set of common registers in the PCI compatible region (0 to 3Fh). The diagram below shows these common registers and their relative position in the configuration space.

A picture containing chart

Description automatically generated

Figure – 6

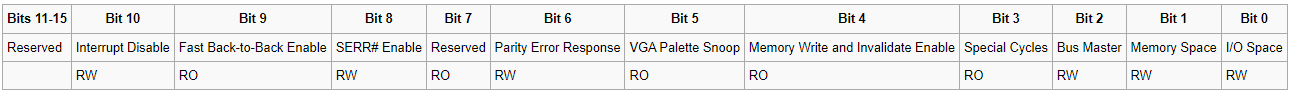
The *device ID* and *vendor ID* are read-only registers that uniquely identify the function. The *vendor ID* is assigned by the [PCI-SIG](https://pcisig.com/) and is different for each vendor, but the device ID is set by the vendor to identify the function. The revision ID field is also set by the vendor to identify hardware and/or firmware versions.

## Command register: -

By writing to “Command Register”, the system controls the device, for example allowing the device to access PCI I/O memory.

Command register has Master Enable (bit 2), if this bit is set then an endpoint may issue memory and I/O read and write requests. When a 0 is written to this register, the device is disconnected from the PCI bus for all accesses except Configuration Space access.

Layout of command register:

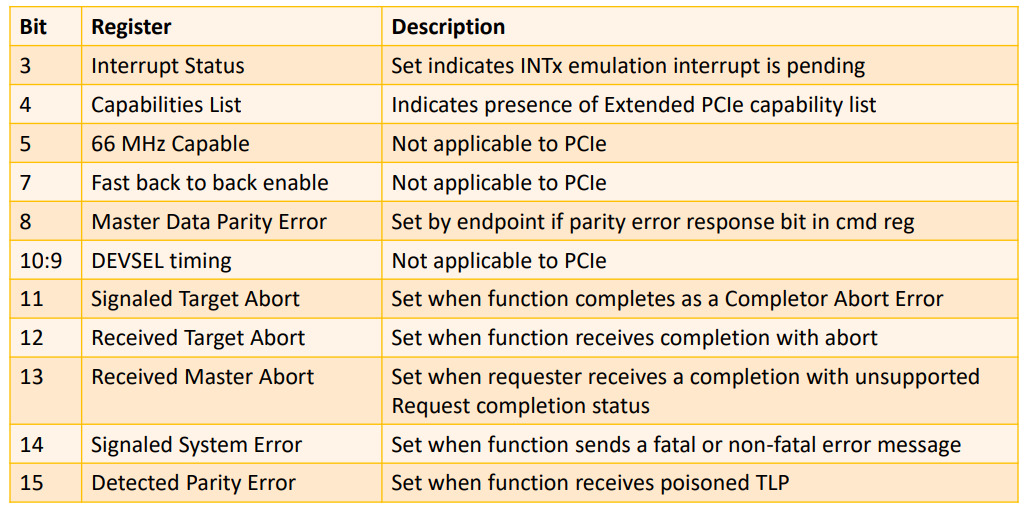


* *Interrupt Disable* - If set to 1 the assertion of the devices INTx# signal is disabled; otherwise, assertion of the signal is enabled.
* *Fast Back-Back Enable* - If set to 1 indicates a device is allowed to generate fast back-to-back transactions; otherwise, fast back-to-back transactions are only 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 has to enable bits 0 and 1 for it to activate.

## Status Register: -

This field gives the status of the device with the meaning of the bits of this field set by the standard.



Layout of Status Register:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 15 | Bit 14 | Bit 13 | Bit 12 | Bit 11 | Bit 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 | 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.

In progress - https://wiki.osdev.org/PCI#Command\_Register

## Class Code register: -

The Class Code register identifies the type of function, with different numbers representing different classes of functionality. For example, a class code of 02h is a network controller or 01h is a mass storage device. These are defined in the PCI Code and ID Assignment Specification*.*

Few class codes are mentioned below.

|  |  |  |
| --- | --- | --- |
| Class Code | Subclass | Prog IF |
| 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 | -- |

For other class codes, refer URL - https://wiki.osdev.org/PCI#Command\_Register

## Header Type register: -

The header type register identifies whether the space is type 0 or type 1. Header Type field is a byte in a register at offset address – 0x0C. Header type field identifies the layout of rest of the header starting at offset position – 0x10.

If bit 7 of this register is set, the device has multiple functions; otherwise, it is a single function device.

|  |  |
| --- | --- |
| Value in “Header Type” field | Meaning |
| 0x0 | General device |
| 0x1 | PCI-to-PCI bridge |
| 0x2 | Card-bus bridge |

### 5.4.1 Header Type 0x0

Below table (Figure – xx) describes Header if the Header Type = 0x0

Graphical user interface, application

Description automatically generated with medium confidence

Figure – PCI configuration header if Header Type = 0x0

Table

Description automatically generated

Figure – PCI configuration header if Header Type = 0x0

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

Below table (Figure – yy)  is applicable if the Header Type is 0x1 (PCI-to-PCI bridge).

Graphical user interface, text, application, email

Description automatically generated

Figure – yy:

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

Below table (Figure – zz) is applicable if the Header Type is 0x2 (PCI-to-CardBus bridge).

Graphical user interface, text, application, email

Description automatically generated

Figure – zz:

## BIST register: -

The BIST register allows control of any built-in-self-test of the function. Bit 7 indicates whether a BIST capability is available, and bit 6 is written to 1 to start the test if available. A result is returned in bits 0 to 3.

## Interrupt Line register: -

The Interrupt Line register is a read-write register that is programmed by the operating system if an interrupt pin is implemented for interrupt routing. The device doesn’t use this value but must provide the register if an interrupt pin is implemented. The Interrupt Pin register is read-only that indicates which legacy interrupts are used (if any). Valid values are 1, 2, 3, and 4 for each of the INTA to INTD legacy interrupt messages.

## Capability Pointer Register: -

Capsbility Pointer Register is at offset 0x34 in configuration space.

The capabilities pointer register indicates an offset, beyond the header registers to further capability register structures. In other words, beyond the PCI registers, the location of other capabilities is not fixed within the configurations space. Instead, capabilities are arranged as a linked list of structures.

The capability pointer register gives the offset for the first capability structure. A popular value would be 40h, the first offset beyond the PCI compatible space - but it need not be.

## BAR Registers: -

* BAR gives the information about address space needed by the device. Base Address Registers (or BARs) can be used to hold memory addresses used by the device, or offsets for port addresses.
* After a device is enumerated, each BAR register will be programmed by physical adress, by mapping this physical address to virtual memory map, PCI device communication will begin.
* Each non-bridge PCI device or Endpoint can implement up to 6 BARs, each of which can respond to different addresses in I/O port and memory-mapped address space. Each BAR describes a region that is between 16 bytes and 2 gigabytes in size, located below 4 gigabyte address space limits. If a platform supports the "Above 4G" option in system firmware, 64-bit BARs can be used.
* Each BAR is 32 bits. 2 BARs are needed for 64-bit BARs (otherwise its not possible to map that device beyond the 4GB boundary).
* **Each BAR corresponds to an address range that serves as a separate communication channel to the PCI device.**

Several Linux kernel PCI functions take the BAR as a parameter to identify which communication channel is to be used, e.g.:

mmio = pci\_iomap(pdev, BAR, pci\_resource\_len(pdev, BAR));

pci\_resource\_flags(dev, BAR);

pci\_resource\_start(pdev, BAR);

pci\_resource\_end(pdev, BAR);

* The memory region that BAR maps is inside PCIe device. After mapping, software(e.g., driver) can read/write the device storage through the mapped memory region.
* BAR is record of the device address starting at memory.

Example: -

# lspci -s 00:04.0 -x

00:04.0 USB controller: Intel Corporation 82801DB/DBM (ICH4/ICH4-M) USB2 EHCI Controller (rev 10)

00: 86 80 cd 24 06 00 00 00 10 20 03 0c 10 00 00 00

10: 00 10 02 f3 00 00 00 00 00 00 00 00 00 00 00 00

20: 00 00 00 00 00 00 00 00 00 00 00 00 f4 1a 00 11

30: 00 00 00 00 00 00 00 00 00 00 00 00 05 04 00 00

We know that offset address of BAR0 register is 0x10, so as per this content of BAR0 register

be – 0xf3021000.

* Bit 0 of BAR register decides, it is I/O space BAR or Memory space layout.
* Memory space BAR layout: -

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

Bit description for Memory space BAR layout: -

|  |  |  |
| --- | --- | --- |
| Bits | Description | Values |
| 0 | Region type | 0 – Memory  1 – I/O |
| 2 - 1 | Type | 0 – Base register is 32 bits wide  1 - Less than 1MB (16 bit)  2 -  Base register is 64-bits wide |
| 3 | Prefetchable | 0 – No  1 - Yes |
| 31 - 4 | Base address | 16-Byte Aligned Base Address |

Out of 32 – bits, the first 4 bit 3:0 are always Read Only.

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).

* I/O space BAR layout: -

|  |  |  |
| --- | --- | --- |
| Bits 31-2 | Bit 1 | Bit 0 |
| 4-Byte Aligned Base Address | Reserved | Always 0 |

### 5.8.1. Address and size of BAR: -

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

To determine the amount of address space needed by a PCI device, we 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 we 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 we can only modify the upper 8-bits.

BAR size must be a power of two (e.g., 1 KiB, 2 MiB), and each area must be aligned in memory such that the lower log2(size) bits of the base address are always zero. For example, assume that an endpoint has a 4 KiB memory area, which gives an address range of 0-0xfff. The host may remap the start of this area to 0x1000 or 0xabcd000 by writing to the BAR register, but not to 0x1080 or 0xabcd100.

When the BAR register is written, the endpoint will ignore LSBs and always return zeros on read. Thus, writing 0xffffffff to the register and then reading back the value indicates the size of the area. For the 4 KiB example this returns 0xfffff00X (the lower four bits are reserved, see specification). To determine the size:

* Clear the lower four bits to zeros (0xfffff000)
* Invert all 32 bits (0xfff)
* Add one to the result (0x1000 = 4096 bytes)

This works for 64-bit areas as well. The value of the next base address register forms the MSBs of the base address. This is described in section 6.2.5.1 of the PCI 3.0 specification.

BAR register reading and size determination is done by kernel, using below function.

File – drivers/pci/probe.c

/\*\*

\* \_\_pci\_read\_base - Read a PCI BAR

\* @dev: the PCI device

\* @type: type of the BAR

\* @res: resource buffer to be filled in

\* @pos: BAR position in the config space

\*

\* Returns 1 if the BAR is 64-bit, or 0 if 32-bit.

\*/

int \_\_pci\_read\_base(struct pci\_dev \*dev, enum pci\_bar\_type type,

struct resource \*res, unsigned int pos)

{

u32 l = 0, sz = 0, mask;

u64 l64, sz64, mask64;

u16 orig\_cmd;

struct pci\_bus\_region region, inverted\_region;

mask = type ? PCI\_ROM\_ADDRESS\_MASK : ~0;

/\* No printks while decoding is disabled! \*/

if (!dev->mmio\_always\_on) {

pci\_read\_config\_word(dev, PCI\_COMMAND, &orig\_cmd);

if (orig\_cmd & PCI\_COMMAND\_DECODE\_ENABLE) {

pci\_write\_config\_word(dev, PCI\_COMMAND,

orig\_cmd & ~PCI\_COMMAND\_DECODE\_ENABLE);

}

}

res->name = pci\_name(dev);

pci\_read\_config\_dword(dev, pos, &l);

pci\_write\_config\_dword(dev, pos, l | mask);

pci\_read\_config\_dword(dev, pos, &sz);

pci\_write\_config\_dword(dev, pos, l);

/\*

\* All bits set in sz means the device isn't working properly.

\* If the BAR isn't implemented, all bits must be 0. If it's a

\* memory BAR or a ROM, bit 0 must be clear; if it's an io BAR, bit

\* 1 must be clear.

\*/

if (sz == 0xffffffff)

sz = 0;

/\*

\* I don't know how l can have all bits set. Copied from old code.

\* Maybe it fixes a bug on some ancient platform.

\*/

if (l == 0xffffffff)

l = 0;

if (type == pci\_bar\_unknown) {

res->flags = decode\_bar(dev, l);

res->flags |= IORESOURCE\_SIZEALIGN;

if (res->flags & IORESOURCE\_IO) {

l64 = l & PCI\_BASE\_ADDRESS\_IO\_MASK;

sz64 = sz & PCI\_BASE\_ADDRESS\_IO\_MASK;

mask64 = PCI\_BASE\_ADDRESS\_IO\_MASK & (u32)IO\_SPACE\_LIMIT;

} else {

l64 = l & PCI\_BASE\_ADDRESS\_MEM\_MASK;

sz64 = sz & PCI\_BASE\_ADDRESS\_MEM\_MASK;

mask64 = (u32)PCI\_BASE\_ADDRESS\_MEM\_MASK;

}

} else {

if (l & PCI\_ROM\_ADDRESS\_ENABLE)

res->flags |= IORESOURCE\_ROM\_ENABLE;

l64 = l & PCI\_ROM\_ADDRESS\_MASK;

sz64 = sz & PCI\_ROM\_ADDRESS\_MASK;

mask64 = PCI\_ROM\_ADDRESS\_MASK;

}

if (res->flags & IORESOURCE\_MEM\_64) {

pci\_read\_config\_dword(dev, pos + 4, &l);

pci\_write\_config\_dword(dev, pos + 4, ~0);

pci\_read\_config\_dword(dev, pos + 4, &sz);

pci\_write\_config\_dword(dev, pos + 4, l);

l64 |= ((u64)l << 32);

sz64 |= ((u64)sz << 32);

mask64 |= ((u64)~0 << 32);

}

if (!dev->mmio\_always\_on && (orig\_cmd & PCI\_COMMAND\_DECODE\_ENABLE))

pci\_write\_config\_word(dev, PCI\_COMMAND, orig\_cmd);

if (!sz64)

goto fail;

sz64 = pci\_size(l64, sz64, mask64);

if (!sz64) {

pci\_info(dev, FW\_BUG "reg 0x%x: invalid BAR (can't size)\n", pos);

goto fail;

}

if (res->flags & IORESOURCE\_MEM\_64) {

if ((sizeof(pci\_bus\_addr\_t) < 8 || sizeof(resource\_size\_t) < 8)

&& sz64 > 0x100000000ULL) {

res->flags |= IORESOURCE\_UNSET | IORESOURCE\_DISABLED;

res->start = 0;

res->end = 0;

pci\_err(dev, "reg 0x%x: can't handle BAR larger than 4GB (size %#010llx)\n",

pos, (unsigned long long)sz64);

goto out;

}

/\*

\* If "A" is a BAR value (a bus address), "bus\_to\_resource(A)" is

\* the corresponding resource address (the physical address used by

\* the CPU. Converting that resource address back to a bus address

\* should yield the original BAR value:

\*

\* resource\_to\_bus(bus\_to\_resource(A)) == A

\*

\* If it doesn't, CPU accesses to "bus\_to\_resource(A)" will not

\* be claimed by the device.

\*/

if ((sizeof(pci\_bus\_addr\_t) < 8) && l) {

/\* Above 32-bit boundary; try to reallocate \*/

res->flags |= IORESOURCE\_UNSET;

res->start = 0;

res->end = sz64 - 1;

pci\_info(dev, "reg 0x%x: can't handle BAR above 4GB (bus address %#010llx)\n",

pos, (unsigned long long)l64);

goto out;

}

}

region.start = l64;

region.end = l64 + sz64 - 1;

pcibios\_bus\_to\_resource(dev->bus, res, &region);

pcibios\_resource\_to\_bus(dev->bus, &inverted\_region, res);

/\*

\* If "A" is a BAR value (a bus address), "bus\_to\_resource(A)" is

\* the corresponding resource address (the physical address used by

\* the CPU. Converting that resource address back to a bus address

\* should yield the original BAR value:

\*

\* resource\_to\_bus(bus\_to\_resource(A)) == A

\*

\* If it doesn't, CPU accesses to "bus\_to\_resource(A)" will not

\* be claimed by the device.

\*/

if (inverted\_region.start != region.start) {

res->flags |= IORESOURCE\_UNSET;

res->start = 0;

res->end = region.end - region.start;

pci\_info(dev, "reg 0x%x: initial BAR value %#010llx invalid\n",

pos, (unsigned long long)region.start);

}

goto out;

fail:

res->flags = 0;

out:

if (res->flags)

pci\_info(dev, "reg 0x%x: %pR\n", pos, res);

return (res->flags & IORESOURCE\_MEM\_64) ? 1 : 0;

}

**APIs/macro functions to access BAR registers: -**

/\*

\* These helpers provide future and backwards compatibility

\* for accessing popular PCI BAR info

\*/

# define pci\_resource\_start(dev, bar) ((dev)->resource[(bar)].start)

# define pci\_resource\_end(dev, bar) ((dev)->resource[(bar)].end)

# define pci\_resource\_flags(dev, bar) ((dev)->resource[(bar)].flags)

# define pci\_resource\_len(dev,bar) \

((pci\_resource\_start((dev), (bar)) == 0 && \

pci\_resource\_end((dev), (bar)) == \

pci\_resource\_start((dev), (bar))) ? 0 : \

\

(pci\_resource\_end((dev), (bar)) - \

pci\_resource\_start((dev), (bar)) + 1))

BAR gives us physical address, to map it to virtual address, we can use pci\_iomap() API.

Signature of this API is below.

void \_\_iomem \*pci\_iomap(struct pci\_dev \*dev, int bar, unsigned long maxlen);

Function description/args:

@dev: PCI device that owns the BAR

@bar: BAR number

@maxlen: length of the memory to map

@maxlen specifies the maximum length to map. If you want to get access to the complete BAR without checking for its length first, pass %0 here.

## Memory-mapped I/O (MMIO)

Memory-mapped I/O uses the same address bus to address both memory and I/O devices – the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. **Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device**, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory.

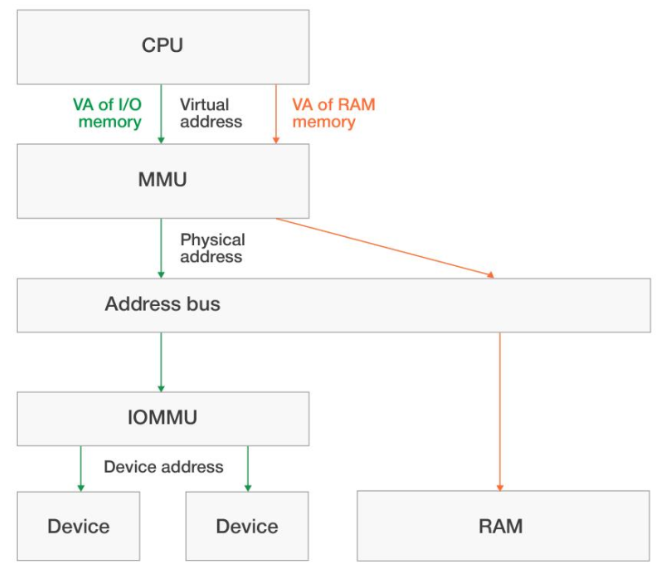


Figure - cc

# Accessing configuration space: -

For access to the PCI configuration space, CPU do not have any such mechanism. This task is usually performed by the Host to PCI Bridge (Host Bridge). There are mechanisms by which software generates configuration access.

Configuration reads and writes can be initiated from the CPU in two ways: one legacy method via I/O addresses 0xCF8 and 0xCFC, and another called memory-mapped configuration.

## Configuration space access in legacy PCI device

Legacy method of configuration access is performed by I/O addresses 0xCF8 and 0xCFC. The legacy method was present in the original PCI, and it is called Configuration Access Mechanism (CAM). It allows for 256 bytes of a device's address space to be reached indirectly via two 32-bit registers called PCI CONFIG\_ADDRESS and PCI CONFIG\_DATA. These registers are at addresses 0xCF8 and 0xCFC in the x86 I/O address space. For example, a software driver (firmware, OS kernel or kernel driver) can use these registers to configure a PCI device by writing the address of the device's register into CONFIG\_ADDRESS, and by putting the data that is supposed to be written to the device into CONFIG\_DATA.

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 us to choose each of the 64 32-bit words. If we 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 | Reserved |

 Register Offset has to point to consecutive DWORDs, ie. bits 1:0 are always 0b00.

The following code segment reads 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);

}

The format of CONFIG\_ADDRESS is the following:

0x80000000 | bus << 16 | device << 11 | function << 8 | offset

As explained previously, addressing a device via Bus, Device, and Function (BDF) is also referred to as "addressing a device geographically." See arch/x86/pci/early.c in the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel) code for an example of code that uses geographical addressing.

When extended configuration space is used on some AMD CPUs, the extra bits 11:8 of the offset are written to bits 27:24 of the CONFIG\_ADDRESS register:

0x80000000 | (offset & 0xf00) << 16 | bus << 16 | device << 11 | function << 8 | (offset & 0xff)

The second method was created for PCI Express. It is called Enhanced Configuration Access Mechanism (ECAM). It extends device's configuration space to 4 KB, with the bottom 256 bytes overlapping the original (legacy) configuration space in PCI. The section of the addressable space is "stolen" so that the accesses from the CPU don't go to memory but rather reach a given device in the PCI Express fabric. During system initialization, BIOS determines the base address for this "stolen" address region and communicates it to the root complex and to the operating system.

Each device has its own 4 KB space and each device's info is accessible through a simple array dev[bus][device][function] so that 256 MB of physical contiguous space is "stolen" for this use (256 buses × 32 devices × 8 functions × 4 KB = 256 MB). The base physical address of this array is not specified. For example, on modern x86 systems the ACPI tables contain the necessary information.

## Configuration space access in PCIe device

The PCI Express bus extends the [Configuration Space](https://wiki.osdev.org/PCI#Configuration_Space) from 256 bytes to 4096 bytes. This extended configuration space cannot be accessed using the legacy PCI method (through ports 0xCF8 and 0xCFC). Instead, an [Enhanced Configuration Mechanism](https://wiki.osdev.org/PCI_Express#Enhanced_Configuration_Mechanism) is provided. However, the legacy configuration space for PCIe devices can still be accessed using the latter.

### Enhanced Configuration Mechanism

The enhanced configuration mechanism makes use of memory mapped address space range/s to access PCI configuration space. The memory address determines the segment group, bus, device, function and register being accessed. On x86 and x64 platforms, the address of each memory area is determined by the ACPI 'MCFG' table. The format of this ACPI table is:

A picture containing application

Description automatically generated

Non-x86 system:

For non-x86 systems the method varies, systems provide themselves with a Device Tree which is parsed at runtime.

To access a specific register within a device's PCI configuration space, we have to use the device's PCI Segment Group and bus to determine which memory mapped PCI configuration space area to use and obtain the starting physical address and starting bus number for that memory mapped area. Once you have the correct starting physical address and starting bus number for that memory mapped area, we would use the following formula to determine where the (4096-byte) area for a function's PCI configuration space is:

|  |
| --- |
| Physical\_Address = MMIO\_Starting\_Physical\_Address + ((Bus - MMIO\_Starting\_Bus) << 20 | Device << 15 | Function << 12). |

Note that it may be a good idea to determine "physical address for this function's PCI configuration space" as part of PCI enumeration and store this physical address alongside any other information you're using to manage PCI devices and drivers (e.g. in your "device manager's" hierarchical tree of device info).

Also note that for absolute maximums (with 65536 PCI segment groups and 256 bus segments per segment group), the amount of physical address space consumed by memory mapped PCI configuration space ranges would be (up to) 16 TiB (or 244 bytes); and ACPI's "MCFG" table may (in theory) be slightly larger than 256 MiB (a 16-byte entry for each individual PCI bus within each PCI segment group plus the 36-byte table header).

Accessing a specific device within the space can be as follows: (((bus \* 256) + (slot \* 8) + func) \* 4096) + offs (each device descriptor is 4096 bytes or 4K long), therefore the former can be interpreted as an array of the type: ((pcie\_ecam[bus][slot][func] \* 4096) + offs).

Finding devices can be done the same as PCI, with the difference that the kernel accesses a memory region rather than using CPU I/O.

[Please undrstand para – 6.2.1 and correct it accordingly.]

***In progress:*** [***https://wiki.osdev.org/PCI***](https://wiki.osdev.org/PCI) ***{page No - 9}***

<https://www.linkedin.com/pulse/pci-express-primer-4-configuration-space-simon-southwell/>

**Very good use this: -** https://en.wikipedia.org/wiki/PCI\_configuration\_space