WIKIPEDIA

PCI configuration space

PCI configuration space is the underlying way that the Conventional PCI, PCI-X and PCI Express perform auto configuration of the cards inserted into their bus.

Contents

Overview

PCI devices have a set of registers referred to as *configuration space* and PCI Express introduces *extended configuration space* for devices. Configuration space registers are mapped to memory locations. Device drivers and diagnostic software must have access to the configuration space, and operating systems typically use APIs to allow access to device configuration space. When the operating system does not have access methods defined or APIs for memory mapped configuration space requests, the driver or diagnostic software has the burden to access the configuration space in a manner that is compatible with the operating system's underlying access rules. In all systems, device drivers are encouraged to use APIs provided by the operating system to access the configuration space of the device.

Technical information

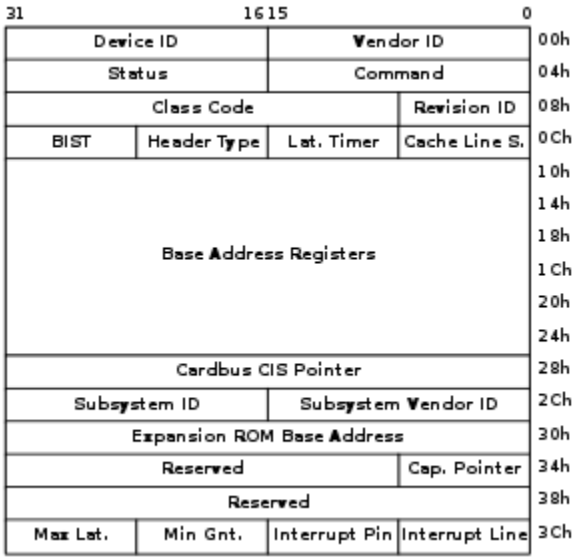
One of the major improvements the PCI Local Bus had over other I/O architectures was its configuration mechanism. In addition to the normal memory-mapped and I/O port spaces, each device function on the bus has a *configuration space*, which is 256 bytes long, addressable by knowing the eight-bit PCI bus, five-bit device, and three-bit function numbers for the device (commonly referred to as the *BDF* or *B/D/F*, as abbreviated from *bus/device/function*). This allows up to 256 buses, each with up to 32 devices, each supporting eight functions. A single PCI expansion card can respond as a device and must implement at least function number zero. The first 64 bytes of configuration space are standardized; the remainder are available for vendor-defined purposes.

In order to allow more parts of configuration space to be standardized without conflicting with existing uses, there can be a list of *capabilities* defined within the first 192 bytes of PCI configuration space. Each capability has one byte that describes which capability it is, and one byte to point to the next capability. The number of additional bytes depends on the capability ID. If capabilities are being used, a bit in the *Status* register is set, and a pointer to the first in a linked list of capabilities is provided in the *Cap. pointer* register defined in the Standardized Registers.

PCI-X 2.0 introduced an 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), 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.

Standardized registers

The *Device ID (DID)* and *Vendor ID (VID)* registers identify the device (such as an IC), and are commonly called the *PCI ID*. The 16-bit vendor ID is allocated by the PCI-SIG. The 16-bit device ID is then assigned by the vendor. There is an ongoing project to collect all known Vendor and Device IDs. (See the external links below.)



Standard registers of PCI Type 0 (Non-Bridge) Configuration Space Header

The *Status* register is used to report which features are supported and whether certain kinds of errors have occurred. The *Command* register contains a bitmask of features that can be individually enabled and disabled. The *Header Type* register values determine the different layouts of remaining 48 bytes (64-16) of the header, depending on the function of the device. That is, Type 1 headers for Root Complex, switches, and bridges. Then Type 0 for endpoints. The *Cache Line Size* register must be programmed before the device is told it can use the memory-write-and-invalidate transaction. This should normally match the CPU's cache line size, but the correct setting is system dependent. This register does not apply to PCI Express.

The *Subsystem ID (SSID)* and the *Subsystem Vendor ID (SVID)* differentiate specific model (such as an add-in card). While the Vendor ID is that of the chipset manufacturer, the Subsystem Vendor ID is that of the card manufacturer. The Subsystem ID is assigned by the subsystem vendor from the same number space as the Device ID. As an example, in the case of wireless network cards, the chipset manufacturer might be Broadcom or Atheros, and the card manufacturer might be Netgear or D-Link. Generally, the Vendor ID–Device ID combination designates which driver the host should load in order to handle the device, as all cards with the same VID:DID combination can be handled by the same driver. The Subsystem Vendor ID–Subsystem ID combination identifies the card, which is the kind of information the driver may use

to apply a minor card-specific change in its operation.

Bus enumeration

To address a PCI device, it must be enabled by being mapped into the system's I/O port address space or memory-mapped address space. The system's firmware, device drivers or the operating system program the *Base Address Registers* (commonly called BARs) to inform the device of its address mapping by writing configuration commands to the PCI controller. Because all PCI devices are in an *inactive* state upon system reset, they will have no addresses assigned to them by which the operating system or device drivers can communicate with them. Either the BIOS or the operating system geographically addresses the PCI slots (for example, the first PCI slot, the second PCI slot, or the third PCI slot, etc., on the motherboard) through the PCI controller using the per slot IDSEL (Initialization Device Select) signals.

Since there is no direct method for the BIOS or operating system to determine which PCI slots have devices installed (nor to determine which functions the device implements) the PCI bus(es) must be *enumerated*. Bus enumeration is performed by attempting to read the vendor ID and device ID (VID/DID) register for each combination of bus number and device number at the device's function #0. Note that device number, different from DID, is merely a device's sequential number on that bus. Moreover, after a new bridge is detected, a new bus number is defined, and device enumeration restarts at device number zero.

PCI BAR Bits

Bits	Description	Values
For all PCI BARs		
0	Region Type	0 = Memory 1 = I/O (deprecated)
For Memory BARs		
2-1	Locatable	0 = any 32-bit 1 = < 1 MiB 2 = any 64-bit
3	Prefetchable	0 = no 1 = yes
31-4	Base Address	16-byte aligned
For I/O BARs (Deprecated)		
1	Reserved	
31-2	Base Address	4-byte aligned

If no response is received from the device's function #0, the bus master performs an abort and returns an all-bits-on value (FFFFFFFF in hexadecimal), which is an invalid VID/DID value, thus a device driver can tell that the specified combination bus/device_number/function (B/D/F) is not present. So, when a read to a function ID of zero for a given bus/device causes the master (initiator) to abort, it must then be presumed that no working device exists on that bus because devices are required to implement function number zero. In this case, reads to the remaining functions numbers (1–7) are not necessary as they also will not exist.

When a read to a specified B/D/F combination for the vendor ID register succeeds, a device driver knows that it exists; it writes all ones to its BARs and reads back the device's requested memory size in an encoded form. The design implies that all address space sizes are a power of two and are naturally aligned.[1]

At this point, the BIOS or operating system will program the memory-mapped and I/O port addresses into the device's BAR configuration register. These addresses stay valid as long as the system remains turned on. Upon power-off, all these settings are lost and the procedure is repeated next time the system is powered back on. Since this entire process is fully automated, the user is spared the task of configuring any newly added hardware manually by changing DIP switches on the cards themselves. This automatic device discovery and address space assignment is how plug and play is implemented.

If a PCI-to-PCI bridge is found, the system must assign the secondary PCI bus beyond the bridge a bus number other than zero, and then enumerate the devices on that secondary bus. If more PCI bridges are found, the discovery continues recursively until all possible domain/bus/device combinations are scanned.

Each non-bridge PCI device function 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:[2][1]

A PCI device can also have an *option ROM* which can contain driver code or configuration information.

Hardware implementation

When performing a *Configuration Space* access, a PCI device does not decode the address to determine if it should respond, but instead looks at the *Initialization Device Select* signal (IDSEL). There is a system-wide unique activation method for each IDSEL signal. The PCI device is required to decode only the lowest order 11 bits of the address space (AD[10] to AD[0]) address/data signals, and can ignore decoding the 21 high order A/D signals (AD[31] to AD[11]) because a *Configuration Space* access implementation has each slot's IDSEL pin connected to a different high order address/data line AD[11] through AD[31]. The IDSEL signal is an actually different pin for each PCI device/adapter slot.

To configure the card in slot *n*, the PCI bus bridge performs a configuration-space access cycle with the PCI device's register to be addressed on lines AD[7:2] (AD[1:0] are always zero since registers are double words (32-bits)), and the PCI function number specified on bits AD[10:8], with all higher-order bits zeros except for AD[*n*+11] being used as the IDSEL signal on a given slot.

To reduce electrically loading down the timing critical (and thus electrically loading sensitive) AD[] bus, the IDSEL signal on the PCI slot connector is usually connected to its assigned AD[*n*+11] pin through a resistor. This causes the PCI's IDSEL signal to reach its active condition more slowly than other PCI bus signals (due to the RC time constant of both the resistor and the IDSEL pin's input capacitance). Thus *Configuration Space* accesses are performed more slowly to allow time for the IDSEL signal to reach a valid level.

The scanning on the bus is performed on the Intel platform by accessing two defined standardized ports. These ports are the *Configuration Space Address* (`0xCF8`) I/O port and *Configuration Space Data* (`0xCFC`) I/O port. The value written to the *Configuration Space Address* I/O port is created by combining B/D/F values and the registers address value into a 32-bit word.

Software implementation

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.[3]

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.[4] 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. Since this process requires a write to a register in order to write the device's register, it is referred to as "indirection".

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 code for an example of code that uses geographical addressing.[5]

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:[6][7]

```
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 4k, 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, firmware determines the base address for this “stolen” address region and communicates it to the root complex and to the operating system. This communication method is implementation-specific, and not defined in the PCI Express specification.

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

See also

- PCI
- PC card
- Root complex

References

1. "Base Address Registers" (http://wiki.osdev.org/PCI#Base_Address_Registers). *PCI*. osdev.org. 2013-12-24. Retrieved 2014-04-17.
2. "PCI configuration methods" (http://www.cs.ucla.edu/~kohler/class/aos-f04/ref/hardware/vgadoc/PCI.TXT). cs.ucla.edu. 2011-11-22. Retrieved 2014-04-17.
3. "Accessing PCI Express* Configuration Registers Using Intel® Chipsets" (http://www.csit-sun.

pub.ro/~cpop/Documentatie_SMP/Intel_Microprocessor_Systems/Intel_ProcessorNew/Intel%20White%20Paper/Accessing%20PCI%20Express%20Configuration%20Registers%20Using%20Intel%20Chipsets.pdf) (PDF). Intel Corporation. Retrieved 27 September 2018.

4. "PCI Configuration Mechanism #1" (http://wiki.osdev.org/PCI#Configuration_Mechanism_.231). *osdev.org*. 2015-01-01. Retrieved 2015-01-01.
5. "kernel/git/stable/linux-stable.git: arch/x86/pci/early.c (Linux kernel stable tree, version 3.12.7)" (https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/arch/x86/pci/early.c?id=refs/tags/v3.12.7). kernel.org. Retrieved 2014-01-10.
6. "kernel/git/stable/linux-stable.git: arch/x86/pci/direct.c (Linux kernel stable tree, version 3.12.7)" (https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/arch/x86/pci/direct.c?id=refs/tags/v3.12.7#n16). kernel.org. Retrieved 2017-09-11.
7. Richter, Robert. "x86: add PCI extended config space access for AMD Barcelona" (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=831d991821daedd4839073dbca55514432ef1768). kernel.org. Retrieved 26 September 2018.
8. "XSDT - OSDev Wiki" (http://wiki.osdev.org/XSDT). Retrieved 2017-04-30.

External links

- PCI Vendor and Device Lists (http://www.pcidatabase.com/index.php)
- The PCI ID Repository (http://pci-ids.ucw.cz/), a project to collect all known IDs
- Description of IO Port usage for PCI configuration (https://pdos.csail.mit.edu/6.828/2005/readings/hardware/vgadoc/PCI.TXT)
- Linux kernel header file with configuration space register definitions (http://lxr.free-electrons.com/source/include/uapi/linux/pci_regs.h)
