Bus Details A *bus* connects components in a computer system. There are several buses in most computers: **PCI** (today = PCIe or *PCI express*; PCI-X is *not* "PCI express, but an older PCI standard), ISA (an even older standard at 8 MHz), AGP (*advanced graphics port*; used for graphics cards before PCIe became common), and the **FSB (*front side bus*)**, to name a few.

The FSB is the most important bus to consider when you are talking about the performance of a computer. It connects the processors (CPUs) to the system memory. When people talk about the speed of a computer, they mean the ***clock speed***. This is used for the CPU, and also determines the FSB speed (which is some ***multiplier*** value of the CPU speed: 1x, .5x, or .25x, the clock speed depending on the hardware). Note many systems default the FSB to the lowest supported speed, so you may be *underclocking* your system!

> ***Overclocking*** is done through manipulating the CPU *multiplier* and the motherboard's front side bus (FSB) speed, until a maximum stable operating frequency is reached. While the idea is simple, variations in the electrical and physical characteristics of computing systems can complicate the process. Several factors limit how much overclocking is possible: bus dividers, voltages, thermal loads, cooling techniques, and other factors can limit overclocking to 1.1x or 1.2x. Even then, overclocking is likely to reduce the lifespan of the hardware.

The PCI bus connects expansion slots for peripherals as well as media drives (disk, DVD) to the rest of the system. Depending on the version, the top speed is only up to 144 MB/sec. (Many non-standard systems ran PCI up to 800 MHz.) This isn't fast enough for many video applications, so an **AGP** bus was added, which was about 8 times faster (more throughput) for video card use.

> In PCI, each device is identified by a bus number, a device number, and a device function. A given computer might have several PCI buses which might be linked (one bus used to extend another bus, joined through a PCI bridge) or independent (several buses all attached to the CPU), or some combination of the two. Generally, large high-end machines with lots of I/O expansion have more complicated PCI topologies than smaller or cheaper systems.
>
> Each device on a PCI bus is assigned a ***device number*** (or "slot") by the PCI bus controller. To save space, a single physical device in a PCI bus *slot* may have several unrelated functions. Each device exposes one or more numbered ***functions***. For example, many graphics cards offer integrated sound hardware for use with HDMI; typically, the graphics capability will be function 0, the sound will be function 1.

Only one device can use a PCI bus at any given moment, and the bus itself is set to the speed of the slowest device connected to it. This is why high-end machines often had (and still have) multiple independent buses—this allows multiple devices to be active simultaneously.

The newer **PCIe (*PCI Express*)** systems are much faster (up to 16 GB/sec) and PCIe has replaced the older PCI+AGP motherboards. PCIe is full-duplex serial (old PCI and ABP were half-duplex, parallel) and doesn't share the bandwidth. (Note, PCI and its descendants send data in packets through the bus.)

PCIe is a point-to-point architecture rather than a bus architecture. With PCI, all devices (and all hardware slots) on the same bus are electrically connected. In PCIe, there are no connections between devices. Instead, each device is connected solely to the controller.

Each connection between device and controller is regarded as its own bus, or ***lane***. Devices are still assigned numbers, but because there can only be one device on each lane, the device number will always be zero. This approach allows software to treat PCIe as if it were PCI, allowing for easier migration from PCI to PCIe.

This point-to-point topology alleviates the bus contention problem in PCI—since there is no bus sharing, concurrent device activity is possible. (Even in PCIe, contention and bus lock-ups are still possible, but much rarer.) Also, each connection runs at the speed of the device on that connection. So slower devices don't slow down the whole bus.

Another feature of PCIe is that devices can connect using multiple lanes, transmitting data in parallel. For example, an x4 PCIe device uses 4 lanes.

Connecting all the buses and other devices attached to the motherboard together is the job of a ***host bridge***. A host often has several (PCI) busses, all connected to the same host bridge. Most hosts have several host bridges, which in turn are connected. (So there is a hierarchy of PCIe buses.)

PCIe busses can be grouped logically into one or more ***domains***. The number of domains depends on the motherboard design.

The fully qualified name of a PCI device is thus:

`domain:bus:slot.function`

(The "`domain:`" and "`.function`" are often omitted, when there is only one domain on the host, and if the device in a slot only provides one function.)

All the host bridges and the buses attached to them, and the devices attached to those buses, are represented as a device hierarchy; devices thus have a kind of pathname.

The devices and buses are usually grouped into two sub-systems (each with its own host bridge) on the motherboard. The first is the ***northbridge***, also known as the *memory controller hub* (MCH) in Intel systems. The northbridge typically handles communications between the CPU, RAM, AGP (or PCIe video card), and the southbridge. The ***southbridge*** is also known as the I/O Controller Hub (ICH) in Intel systems. It implements the "slower" capabilities of the motherboard including the PCI expansion slots, serial and parallel ports, keyboard, mouse, "on-board" video chip, USB, NICs, and so on. (See IntelArch.gif.)

AMD improved performance in 2003 by connecting memory directly to the CPU. In 2009, Intel eliminated the Northbridge (MCH) completely (see IntelP55.gif).

Devices on busses need a way to get the attention of the bus controller, and the CPU needs a way to get the attention of some device. To enable this, devices are assigned several values:

- **IRQ**s: Only 15 of these (0-15, 2=9), all but a few are reserved. (4=ttyS0, 3=ttyS1, 7=lp0, 5=lp1, 6=fd0) Some devices can share a single IRQ but this is rare. Some systems allow *soft IRQs* so there is no limit on the number of them.

  IRQ are supposed to be assigned automatically on PCI and ISA-PnP, and manually on ISA devices (using jumpers, DIP switches, or special software). Many PnP devices don't work according to the standard and will only work if a particular IRQ is assigned, so manually setting these may be needed.

  For PCI use **`lspci`** [`-t`] and `/proc/bus/pci`. For *jumperless ISA* and ISA-PnP, use `isadump` > `/etc/isapnp.conf` > `isapnp`, `pnpdump` and `pnpprobe`. See also **`/proc/interrupts`** and `/proc/irq/*`.

  Originally, each device had its own private IRQ. Today PCI devices share IRQs. When a device interrupts the CPU an *interrupt handler* checks to see which device actually caused the interrupt, and then runs the proper code. Note `/proc/interrupts` doesn't know which devices share a common IRQ; use the command "`dmesg |grep -i irq`" to learn which IRQ some device uses.

- **I/O Base Address** (sometimes called a *port*, but not related to the network term!) Memory-mapped I/O is a common technique. The CPU reads or writes a certain address, but instead of the data coming from RAM, it comes from or goes to some RAM (or *registers*) on the device. Each device that communicates this way will respond to a certain address or (small) range of addresses only. If PnP doesn't set this correctly, you may have to set it manually or the kernel won't be able to talk to the device (or two devices may interfere with each other.) (Some std. address ranges are: 0x3F8=ttyS0, 0x378–37F=lp0, 0x3F0–0F7=fd0)
- **DMA** (*Direct Memory Access*) is a high-speed transfer of data directly to RAM from some device. Normally software running on the CPU reads a few bytes from the device (to the CPU) and then writes it to RAM, using the I/O port. Going direct without involving the CPU is faster. Like IRQs and I/O addresses, this should normally be configured automatically for those devices that use it.

NDISwrapper is special software that can translate between Windows32 driver API and the Linux API. It was designed for NICs and Wi-Fi cards that lack Linux drivers. You simply install a (legal copy of a) Windows driver (a pair of files, `foo.sys` and `foo.inf`) and use `ndiswrapper` build a kernel module from it that you can load. Since Windows drivers are notorious for bypassing the Win32 API, this won't always work, but is worth a try if you can't find a Linux/Unix driver.

To use:

```
  yum install --enablerepo=livna kmod-
ndiswrapper
  ndiswrapper -i foo.inf   # creates module
  ndiswrapper -m  # adds conf info to
/etc/modprobe.d/ndiswrapper
```

Hardware changes may be auto detected at every boot. On Linux this is done with `kudzu`. Solaris doesn't run auto detection at each boot, and different commands are used to scan for different types of new hardware. For disks that have been added, use `devfsadm(1M)`. For any new hardware you need to perform a *reconfiguration boot*. At the "`ok`" prompt use "`boot -r`", or you can `touch /reconfigure` and reboot normally.

On Linux you can use `lsdev` to view a hardware inventory. On Solaris use the various `prt*` commands (e.g. `prtconfig`), the `sysdef` command, and the various `*adm` commands (e.g. `cfgadm`). The identifying data that devices report are converted to vendor names, device names, types, and models, using a large database called the PCI ID Repository. The local copy of this DB can be refreshed by running the Linux command `update-pciids`; this is a good idea after adding any new hardware to your systems.