Plug-and-Play-HOWTO

Table of Contents

Table of Contents

Table of Contents

Plug-and-Play-HOWTO

David S. Lawyer mailto:dave@lafn.org

v1.15, August 2007

Explains in detail low-level resources such as addresses, interrupts, etc. Covers both the PCI bus, which is inherently Plug and Play (PnP) and PnP on the old ISA bus. If PnP did it's job right, you wouldn't need this howto. But in case it doesn't, or if you have old hardware that doesn't use PnP for all the cards, then this HOWTO should help. It doesn't cover what's called "Universal Plug and Play" (UPnP).

1. Introduction

1.1 1. Copyright, Trademarks, Disclaimer, & Credits

Copyright

Disclaimer

While I haven't intentionally tried to mislead you, there are likely a number of errors in this document. Please let me know about them. Since this is free documentation, it should be obvious that I cannot be held legally responsible for any errors.

Trademarks.

Any brand names (starts with a capital letter such as MS Windows) should be assumed to be a trademark). Such trademarks belong to their respective owners.

Credits

- March 2000: Daniel Scott proofread this and found many typos, etc.
- June 2000: Pete Barrett gave a workaround to prevent Windows from zeroing PCI IRQs.

- August 2004: Ross Boylan found typos, etc. and pointed out lack of clarity in telling the BIOS if it's a PnP OS

1.2 Future Plans; You Can Help

Please let me know of any errors in facts, opinions, logic, spelling, grammar, clarity, links, etc. But first, if the date is over a several months old, check to see that you have the latest version. Please send me any info that you think belongs in this document.

I haven't studied the code used by various Linux drivers and the kernel to implement Plug-and-Play. But I have sampled a little of it (especially some of the comments). Thus this HOWTO is still incomplete. It needs to explain more about "hot swapping", "hot-plug" and about the new PnP software for kernel 2.6. The history of Linux PnP is not well covered. Also, it doesn't cover firewire. It likely has some inaccuracies (let me know where I'm wrong). In this HOWTO I've sometimes used ?? to indicate that I don't really know the answer.

1.3 New Versions of this HOWTO

New versions of the Plug-and-Play-HOWTO should appear every year or so and will be available to browse and/or download at LDP mirror sites. For a list of mirror sites see: http://tldp.org/mirrors.html. Various formats are available. If you only want to quickly check the date of the latest version look at: http://tldp.org/HOWTO/Plug-and-Play-HOWTO.html. The version you are now reading is: v1.15, August 2007 .

1.4 New in Recent Versions

For a full revision history going back to the first version see the source file (in linuxdoc format) at http://cvsview.tldp.org/index.cgi/LDP/howto/linuxdoc/Plug-and-Play-HOWTO.sgml

- v1.15 Aug. 2007 Revised interrupt sections. Removed 2 redundant and confusing paragraphs containing a mystery function "h()"
- v1.14 Feb. 2006: Revised "How Linux Does PnP"; LPC was intended to be config. by the BIOS. Balancing IRQs. Linux can find drivers for detected devices.
- v1.13 July 2005: IRQ conflicts. Better clarity in resource descriptions. /proc/bus. PCI configuration space accessed via IO address space. More hardware detection tools. "Can't allocate region" error message.
- v1.12 March 2005: /dev/eth0 doesn't exist anymore. Info in /sys and /proc changed for kernel 2.6. PCI Config. address space is "geographic". scanpci may find a device that lspci can't. Kernel may assign addresses at boot-time.

1.5 General Introduction. Do you need this HOWTO?

Plug-and-play (PnP) is a system which automatically detects devices such as disks, sound cards, ethernet cards, modems, etc. It finds all devices on the PCI bus and all devices that support PnP on the old ISA bus. Before PnP, many devices were automatically searched for by non-PnP methods, but were sometimes not found. PnP provides a way to find all devices that support PnP. It also does some low-level configuring of them. Non-PnP devices (or PnP devices which have not been correctly PnP-configured), can often be detected by non-PnP methods. The PCI bus is inherently PnP while the old ISA bus originally wasn't PnP but had PnP support added to it later. So sometimes PnP is used to only mean PnP for the old ISA bus. For example, when you see a boot-time message from "isapnp" and it reads: "Plug & Play device" it only means an ISA Plug &

Play device. In this HOWTO, PnP means PnP for both the ISA and the PCI bus.

As time goes by the Linux kernel is became better at supporting PnP. In the late 20th century, one could say that Linux was not really a PnP OS. But the claim is made that with version 2.6 of the kernel, Linux is now fully PnP (provided the kernel is built with appropriate PnP support). While the PnP system is not centralized like it is in MS Windows (with its registry) the decentralized Linux PnP seems to work OK.

Linux does keep track of resource assignments requested by device drivers and refuses any request if it thinks it would cause a conflict. The kernel also provides programs that device drivers can call on to do their own plug-and-play. The kernel also reads all configuration registers of all PnP devices and maintains tables of them that device drivers can consult. This table helps drivers find their hardware. Kernel 2.6 provides better support for "hot plug".

The BIOS hardware of your PC likely does some plug-and-play work too. Thus if everything works OK PnP-wise, you can use your computer without needing to know anything about plug-and-play. But if some devices which are supported by Linux don't work (because they're not discovered or configured correctly by PnP) then you may need to read some of this HOWTO. You'll learn not only about PnP but also learn something about how communication takes place inside the computer. If you have a modern computer with a PCI bus but no ISA bus, you may skip over or skim the parts about the ISA bus.

If you're having problems with a device, watch the messages displayed at boot-time (go back thru them using Shift-PageUp). If this doesn't also display early messages from the BIOS use the "Pause" key. See Pause

Check to see that you have the right driver for a device, and that the driver is being found and used. If the driver is a module, type "lsmod" (as the root user) to see it it's loaded (in use). If it's not a module then it should be built into the kernel.

This HOWTO doesn't cover the problem of finding and installing device drivers. Perhaps it should. One problem is that a certain brand of a card (or other physical device) may not say what kind of chips are used in it. The driver name is often the same as the chip name and not the brand name. One way to start to check on a driver is to see if it is discussed in the kernel documentation, in another HOWTO, or on the Internet. Warning: Such documentation may be out of date.

The PCI bus computers (no ISA bus) have significantly reduced the number of things that can go wrong. For the ISA bus and the lack of kernel support for ISA Pnp (before kernel 2.4), there was much more that could go wrong. Remember that sometimes problems which seem to be PnP related are actually due to defective hardware or to hardware that doesn't fully conform to PnP specs.

2. What PnP Should Do: Allocate "Bus-Resources"

2.1 What is Plug-and-Play (PnP)?

If you don't understand this section, read the next section Hardware Devices and Communication with them

Oversimplified, Plug-and-Play tells the software (device drivers) where to find various pieces of hardware (devices) such as modems, network cards, sound cards, etc. Plug-and-Play's task is to match up physical devices with the software (device drivers) that operates them and to establish channels of communication between each physical device and its driver. In order to achieve this, PnP allocates and sets the following "bus-resources" in hardware: I/O addresses, memory regions, IRQs, DMA channels (LPC and ISA buses only). These 4 things are sometimes called "1st order resources" or just "resources". Pnp maintains a record of

what it's done and allows device drivers to get this information. If you don't understand what these 4 bus-resources are, read the following subsections of this HOWTO: I/O Addresses, IRQs, DMA Channels, Memory Regions. An article in Linux Gazette regarding 3 of these bus-resources is Introduction to IRQs, DMAs and Base Addresses. Once these bus-resources have been assigned (and if the correct driver is installed), the actual driver and the "files" for it in the /dev directory are ready to use.

This PnP assignment of bus-resources is sometimes called "configuring" but it is only a low level type of configuring. The /etc directory has many configuration files but most all of them are not for PnP configuring. So most of the configuring of hardware devices has nothing to do with PnP or bus-resources. For, example the initializing of a modem by an "init string" or setting it's speed is not PnP. Thus when talking about PnP, "configuring" means only a certain type of configuring. While other documentation (such as for MS Windows) simply calls bus-resources "resources", I sometimes use the term "bus-resources" instead of just "resources" so as to distinguish it from the multitude of other kinds of resources.

PnP is a process which is done by various software and hardware. If there was just one program that handled PnP in Linux, it would be simple. But with Linux each device driver does it's own PnP, using software supplied by the kernel. The BIOS hardware of the PC does PnP when a PC is first powered up. And there's a lot more to it than this.

2.2 Hardware Devices and Communication with them

A computer consists of a CPU/processor to do the computing and RAM memory to store programs and data (for fast access). In addition, there are a number of devices such as various kinds of disk-drives, a video card, a keyboard, network devices, modem cards, sound devices, the USB bus, serial and parallel ports, etc. In olden days most devices were on cards inserted into slots in the PC. Today, many devices that were formerly cards, are now on-board since they are contained in chips on the motherboard. There is also a power supply to provide electric energy, various buses on a motherboard to connect the devices to the CPU, and a case to put all this into.

Cards which plug into the motherboard may contain more than one device. Memory chips are also sometimes considered to be devices but are not plug-and-play in the sense used in this HOWTO.

For the computer system to work right, each device must be under the control of its "device driver". This is software which is a part of the operating system (perhaps loaded as a module) and runs on the CPU. Device drivers are associated with "special files" in the /dev directory although they are not really files. They have names such as hda3 (third partition on hard drive a), ttyS1 (the second serial port), eth0 (the first ethernet card), etc.

The eth0 device is for an ethernet card (nic card). Formerly it was /dev/eth0 but it's now just a virtual device in the kernel. What eth0 refers to depends on the type of ethernet card you have. If the driver is a module, this assignment is likely in an internal kernel table but might be found in /etc/modules.conf (called "alias"). For example, if you have an ethernet card that uses the "tulip" chip you could put "alias eth0 tulip" into /etc/modules.conf so that when your computer asks for eth0 it finds the tulip driver. However, modern kernels can usually find the right driver module so that you seldom need to specify it yourself.

To control a device, the CPU (under the control of the device driver) sends commands and data to, and reads status and data from the various devices. In order to do this each device driver must know the address of the device it controls. Knowing such an address is equivalent to setting up a communication channel, even though the physical "channel" is actually the data bus inside the PC which is shared with many other devices.

This communication channel is actually a little more complex than described above. An "address" is actually a range of addresses so that sometimes the word "range" is used instead of "address". There could even be more that one range (with no overlapping) for a single device. Also, there is a reverse part of the channel (known as interrupts) which allows devices to send an urgent "help" request to their device driver.

2.3 Addresses

The PCI bus has 3 address spaces: I/O, main memory (IO memory), and configuration. The old ISA bus lacks a genuine "configuration" address space. Only the I/0 and IO memory spaces are used for device IO. Configuration addresses are fixed and can't be changed so they don't need to be allocated. For more details see PCI Configuration Address Space

When the CPU wants to access a device, it puts the device's address on a major bus of the computer (for PCI: the address/data bus). All types of addresses (such as both I/O and main memory) share the same bus inside the PC. But the presence or absence of voltage on certain dedicated wires in the PC's bus tells which "space" an address is in: I/O, main memory, (see Memory Ranges), or configuration (PCI only). This is a little oversimplified since telling a PCI device that it's a configuration space access is actually more complex than described above. See PCI Configuration Address Space for details. See Address Details for more details on addressing in general.

The addresses of a device are stored in it's registers in the physical device. They can be changed by software and they can be disabled so that the device has no address at all. Except that the PCI configuration address can't be changed or disabled.

2.4 I/O Addresses (principles relevant to other resources too)

Devices were originally located in I/O address space but today they may use space in main memory. An I/0 address is sometimes just called "I/O", "IO", "i/o" or "io". The terms "I/O port" or "I/O range" are also used. Don't confuse these IO ports with "IO memory" located in main memory. There are two main steps to allocate the I/O addresses (or some other bus-resources such as interrupts on the ISA bus):

1. Set the I/O address, etc. in the hardware (in one of its registers)
2. Let its device driver know what this I/O address, etc. is

Often, the device driver does both of these (sort of). The device driver doesn't actually need to set an I/O address if it finds out that the address has been previously set (perhaps by the BIOS) and is willing to accept that address. Once the driver has either found out what address has been previously set or sets the address itself, then it obviously knows what the address is so there is no need to let the driver know the address --it already knows it.

The two step process above (1. Set the address in the hardware. 2. Let the driver know it.) is something like the two part problem of finding someone's house number on a street. Someone must install a number on the front of the house so that it may be found and then people who might want to go to this address must obtain (and write down) this house number so that they can find the house. For computers, the device hardware must first get its address put into a special register in its hardware (put up the house number) and then the device driver must obtain this address (write the house number in its address book). Both of these must be done, either automatically by software or by entering the data manually into configuration files. Problems may occur when only one of them gets done right.

For manual PnP configuration some people make the mistake of doing only one of these two steps and then wonder why the computer can't find the device. For example, they may use "setserial" to assign an address to a serial port without realizing that this only tells the driver an address. It doesn't set the address in the serial port hardware itself. If you told the driver wrong then you're in trouble. Another way to tell the driver is to give the address as an option to a kernel module (device driver). If what you tell it is wrong, there could be problems. A smart driver may detect how the hardware is actually set and reject the incorrect information supplied by the option (or at least issue an error message).

An obvious requirement is that before the device driver can use an address it must be first set in the physical device (such as a card). Since device drivers often start up soon after you start the computer, they sometimes try to access a card (to see if it's there, etc.) before the address has been set in the card by a PnP configuration program. Then you see an error message that they can't find the card even though it's there (but doesn't yet have an address yet).

What was said in the last few paragraphs regarding I/O addresses applies with equal force to most other bus-resources: Memory Ranges, IRQs --Overview and DMA Channels. What these are will be explained in the next 3 sections. The exception is that interrupts on the PCI bus are not set by card registers but are instead routed (mapped) to IRQs by a chip on the motherboard. Then the IRQ a PCI card is routed to is written into the card's register for information purposes only.

To see what IO addresses are used on your PC, look at the /proc/ioports file.

2.5 Memory Ranges

Many devices are assigned address space in main memory. It's sometimes called "shared memory" or "memory-mapped IO" or "IO memory". This memory is physically located inside the physical device but the computer accesses it just like it would access memory on memory chips. When discussing bus-resources it's often just called "memory", "mem", or "iomem". In addition to using such "memory", such a device might also use conventional IO address space. To see what mem is in use on your computer, look at /proc/iomem. This "file" includes the memory used by your ordinary RAM memory chips so it shows memory allocation in general and not just iomem allocation. If you see a strange number instead of a name, it's likely the number of a PCI device which you can verify by typing "lspci".

When you insert a card that uses iomem, you are in effect also inserting a memory module for main memory. A high address is selected for it by PnP so that it doesn't conflict with the main memory modules (chips). This memory can either be ROM (Read Only Memory) or shared memory. Shared memory is shared between the device and the CPU (running the device driver) just as IO address space is shared between the device and the CPU. This shared memory serves as a means of data "transfer" between the device and main memory. It's Input-Output (IO) but it's not done in IO space. Both the card and the device driver need to know the memory range.

ROM (Read Only Memory) on cards is a different kind of iomem. It is likely a program (perhaps a device driver) which will be used with the device. It could be initialization code so that a device driver is still required. Hopefully, it will work with Linux and not just MS Windows. It may need to be shadowed which means that it is copied to your main memory chips in order to run faster. Once it's shadowed it's no longer "read only".

2.6 IRQs --Overview

After reading this you may want to read Interrupts --Details for many more details. The following is intentionally oversimplified: Besides the address, there is also an interrupt number to deal with (such as IRQ 5). It's called an IRQ (Interrupt ReQuest) number or just an "irq" for short. We already mentioned above that the device driver must know the address of a card in order to be able to communicate with it.

But what about communication in the opposite direction? Suppose the device needs to tell its device driver something immediately. For example, the device may be receiving a lot of bytes destined for main memory and its buffer used to store these bytes is almost full. Thus the device needs to tell its driver to fetch these bytes at once before the buffer overflows from the incoming flow of bytes. Another example is to signal the driver that the device has finished sending out a bunch of bytes and is now waiting for some more bytes from the driver so that it can send them too.

How should the device rapidly signal its driver? It may not be able to use the main data bus since it's likely already in use. Instead it puts a voltage on a dedicated interrupt wire (also called line or trace) which is often reserved for that device alone. This voltage signal is called an Interrupt ReQuest (IRQ) or just an "interrupt" for short. There are the equivalent of 16 (or 24, etc.) such wires in a PC and each wire leads (indirectly) to a certain device driver. Each wire has a unique IRQ (Interrupt ReQuest) number. The device must put its interrupt on the correct wire and the device driver must listen for the interrupt on the correct wire. Which wire the device sends such "help requests" on is determined by the IRQ number stored in the device. This same IRQ number must be known to the device driver so that the device driver knows which IRQ line to listen on.

Once the device driver gets the interrupt from the device it must find out why the interrupt was issued and take appropriate action to service the interrupt. On the ISA bus, each device usually needs its own unique IRQ number. For the PCI bus and other special cases, the sharing of IRQs is allowed (two or more PCI devices may have the same IRQ number). Also, for PCI, each PCI device has a fixed "PCI Interrupt" wire. But a programmable routing chip maps the PCI wires to ISA-type interrupts. See Interrupts --Details for details on how all the above works.

2.7 DMA (Direct Memory Access) or Bus Mastering

For the PCI bus, DMA and Bus Mastering mean the same thing. Prior to the PCI bus, Bus Mastering was rare and DMA worked differently and was slow. Direct Memory Access (DMA) is where a device is allowed to take over the main computer bus from the CPU and transfer bytes directly to main memory or to some other device. Normally the CPU would make a transfer from a device to main memory in a two step process:

1. reading a chunk of bytes from the I/O memory space of the device and putting these bytes into CPU itself
2. writing these bytes from the CPU to main memory

With DMA it's a one step process of sending the bytes directly from the device to memory. The device must have DMA capabilities built into its hardware and thus not all devices can do DMA. While DMA is going on, the CPU can't do too much since the main bus is being used by the DMA transfer.

The old ISA bus can do slow DMA while the PCI bus does "DMA" by Bus Mastering. The LPC bus has both the old DMA and the new DMA (bus mastering). On the PCI bus, what more precisely should be called "bus mastering" is often called "Ultra DMA", "BM-DNA", "udma", or just "DMA", Bus mastering allows devices to temporarily become bus masters and to transfer bytes almost like the bus master was the CPU. It doesn't use any channel numbers since the organization of the PCI bus is such that the PCI hardware knows which

device is currently the bus master and which device is requesting to become a bus master. Thus there is no resource allocation of DMA channels for the PCI bus and no dma channel resources exist for this bus. The LPC (Low Pin Count) bus is supposed to be configured by the BIOS so users shouldn't need to concern themselves with its DMA channels.

2.8 DMA Channels (not for PCI bus)

This is only for the LPC bus and the old ISA bus. When a device wants to do DMA it issues a DMA-request using dedicated DMA request wires much like an interrupt request. DMA actually could have been handled by using interrupts but this would introduce some delays so it's faster to do it by having a special type of interrupt known as a DMA-request. Like interrupts, DMA-requests are numbered so as to identify which device is making the request. This number is called a DMA-channel. Since DMA transfers all use the main bus (and only one can run at a time) they all actually use the same channel for data flow but the "DMA channel" number serves to identify who is using the "channel". Hardware registers exist on the motherboard which store the current status of each "channel". Thus in order to issue a DMA-request, the device must know its DMA-channel number which must be stored in a special register on the physical device.

2.9 "Resources" for both Device and Driver

Thus device drivers must be "attached" in some way to the hardware they control. This is done by allocating bus-resources (I/O, Memory, IRQ's, DMA's) to both the physical device and letting the device driver to find out about it. For example, a serial port uses only 2 resources: an IRQ and an I/O address. Both of these values must be supplied to the device driver and the physical device. The driver (and its device) is also given a name in the /dev directory (such as ttyS1). The address and IRQ number is stored by the physical device in configuration registers on its card (or in a chip on the motherboard). Old hardware (in the mid 1990's) used switches (or jumpers) to physically set the IRQ and address in the hardware. This setting remained fixed until someone remover the computer's cover and moved the jumpers.

But for the case of PnP (no jumpers), the configuration register data is usually lost when the PC is powered down (turned off) so that the bus-resource data must be supplied to each device anew each time the PC is powered on.

2.10 Resources are Limited

Ideal Computers

The architecture of the PC provides only a limited number of resources: IRQ's, DMA channels, I/O address, and memory regions. If there were only a limited number devices and they all used standardized bus-resources values (such as unique I/O addresses and IRQ numbers) there would be no problem of attaching device drivers to devices. Each device would have a fixed resources which would not conflict with any other device on your computer. No two devices would have the same addresses, there would be no IRQ conflicts on the ISA bus, etc. Each driver would be programmed with the unique addresses, IRQ, etc. hard-coded into the program. Life would be simple.

Another way to prevent address conflicts would be to have each card's slot number included as part of the address. Thus there could be no address conflict between two different cards (since they are in different slots). Card design would not allow address conflicts between different functions of the card. It turns out that the configuration address space (used for resource inquiry and assignment) actually does this. But it's not done for I/O addresses nor memory regions. Sharing IRQs as on the PCI bus also avoids conflicts but may cause other

problems.

Real Computers

But PC architecture has conflict problems. The increase in the number of devices (including multiple devices of the same type) has tended to increase potential conflicts. At the same time, the introduction of the PCI bus, where two or more devices can share the same interrupt and the introduction of more interrupts, has tended to reduce conflicts. The overall result, due to going to PCI, has been a reduction in conflicts since the scarcest resource is IRQs. However, even on the PCI bus it's more efficient to avoid IRQ sharing. In some cases where interrupts happen in rapid succession and must be acted on fast (like audio) sharing can cause degradation in performance. So it's not good to assign all PCI devices the same IRQ, the assignment needs to be balanced. Yet some people find that all their PCI devices are on the same IRQ.

So devices need to have some flexibility so that they can be set to whatever address, IRQ, etc. is needed to avoid any conflicts and achieve balancing. But some IRQ's and addresses are pretty standard such as the ones for the clock and keyboard. These don't need such flexibility.

Besides the problem of conflicting allocation of bus-resources, there is a problem of making a mistake in telling the device driver what the bus-resources are. This is more likely to happen for the case of old-fashioned manual configuration where the user types in the resources used into a configuration file stored on the harddrive. This often worked OK when resources were set by jumpers on the cards (provided the user knew how they were set and made no mistakes in typing this data to configuration files). But with resources being set by PnP software, they may not always get set the same and this may mean trouble for any manual configuration where the user types in the values of bus-resources that were set by PnP.

The allocation of bus-resources, if done correctly, establishes non-conflicting channels of communication between physical hardware and their device drivers. For example, if a certain I/O address range (resource) is allocated to both a device driver and a piece of hardware, then this has established a one-way communication channel between them. The driver may send commands and other info to the device. It's actually more than one-way communications since the driver may get information from the device by reading its registers. But the device can't initiate any communication this way. To initiate communication the device needs an IRQ so it can send interrupts to its driver. This creates a two-way communication channel where both the driver and the physical device can initiate communication.

2.11 Second Introduction to PnP

The term Plug-and-Play (PnP) has various meanings. In the broad sense it is just auto-configuration where one just plugs in a device and it configures itself. In the sense used in this HOWTO, PnP means the configuring PnP bus-resources (setting them in the physical devices) and letting the device drivers know about it. For the case of Linux, it is often just a driver determining how the BIOS has set bus-resources and if necessary, the driver giving a command to change (reset) the bus-resources. "PnP" often just means PnP on the ISA bus so that the message from isapnp: "No Plug and Play device found" just means that no ISA PnP devices were found. The standard PCI specifications (which were invented before coining the term "PnP") provide the equivalent of PnP for the PCI bus.

PnP matches up devices with their device drivers and specifies their communication channels (by allocating bus-resources). It electronically communicates with configuration registers located inside the physical devices using a standardized protocol. On the ISA bus before Plug-and-Play, the bus-resources were formerly set in hardware devices by jumpers or switches. Sometimes the bus-resources could be set into the hardware electronically by a driver (usually written only for a MS OS but in rare cases supported by a Linux driver).

This was something like PnP but there was no standardized protocol used so it wasn't really PnP. Some cards had jumper setting which could be overridden by such software. For Linux before PnP, most software drivers were assigned bus-resources by configuration files (or the like) or by probing the for the device at addresses where it was expected to reside. But these methods are still in use today to allow Linux to use old non-PnP hardware. And sometimes these old methods are still used today on PnP hardware (after say the BIOS has assigned resources to hardware by PnP methods).

The PCI bus was PnP-like from the beginning, but it's not usually called PnP or "plug and play" with the result that PnP often means PnP on the ISA bus. But PnP in this documents usually means PnP on either the ISA or PCI bus.

2.12 How Pnp Works (simplified)

Here's how PnP should work in theory. The hypothetical PnP configuration program finds all PnP devices and asks each what bus-resources it needs. Then it checks what bus-resources (IRQs, etc.) it has to give away. Of course, if it has reserved bus-resources used by non-PnP (legacy) devices (if it knows about them) it doesn't give these away. Then it uses some criteria (not specified by PnP specifications) to give out the bus-resources so that there are no conflicts and so that all devices get what they need (if possible). It then indirectly tells each physical device what bus-resources are assigned to it and the devices set themselves up to use only the assigned bus-resources. Then the device drivers somehow find out what bus-resources their devices use and are thus able to communicate effectively with the devices they control.

For example, suppose a card needs one interrupt (IRQ number) and 1 MB of shared memory. The PnP program reads this request from the configuration registers on the card. It then assigns the card IRQ5 and 1 MB of memory addresses space, starting at address 0xe9000000. The PnP program also reads identifying information from the card telling what type of device it is, its ID number, etc. Then it directly or indirectly tells the appropriate device driver what it's done. If it's the driver itself that is doing the PnP, then there's no need to find a driver for the device (since it's driver is already running). Otherwise a suitable device driver needs to be found and sooner or later told how it's device is configured.

It's not always this simple since the card (or routing table for PCI) may specify that it can only use certain IRQ numbers or that the 1 MB of memory must lie within a certain range of addresses. The details are different for the PCI and ISA buses with more complexity on the ISA bus.

One way commonly used to allocate resources is to start with one device and allocate it bus-resources. Then do the same for the next device, etc. Then if finally all devices get allocated resources without conflicts, then all is OK. But if allocating a needed resource would create a conflict, then it's necessary to go back and try to make some changes in previous allocations so as to obtain the needed bus-resource. This is called rebalancing. Linux doesn't do rebalancing but MS Windows does in some cases. For Linux, all this is done by the BIOS and/or kernel and/or device drivers. In Linux, the device driver doesn't get it's final allocation of resources until the driver starts up, so one way to avoid conflicts is just not to start any device that might cause a conflict. However, the BIOS often allocates resources to the physical device before Linux is even booted and the kernel checks PCI devices for addresses conflicts at boot-time.

There are some shortcuts that PnP software may use. One is to keep track of how it assigned bus-resources at the last configuration (when the computer was last used) and reuse this. BIOSs do this as does MS Windows and this but standard Linux doesn't. But in a way it does since it often uses what the BIOS has done. Windows stores this info in its "Registry" on the hard disk and a PnP/PCI BIOS stores it in non-volatile memory in your PC (known as ESCD; see The BIOS's ESCD Database). Some say that not having a registry (like Linux) is better since with Windows, the registry may get corrupted and is difficult to edit. But PnP in Linux has

problems too.

While MS Windows (except for Windows 3.x and NT4) were PnP, Linux was not originally a PnP OS but has been gradually becoming a PnP OS. PnP originally worked for Linux because a PnP BIOS would configure the bus-resources and the device drivers would find out (using programs supplied by the Linux kernel) what the BIOS has done. Today, most drivers can issue commands to do their own bus-resource configuring and don't need to always rely on the BIOS. Unfortunately a driver could grab a bus-resource which another device will need later on. Some device drivers may store the last configuration they used in a configuration file and use it the next time the computer is powered on.

If the device hardware remembered its previous configuration, then there wouldn't be any hardware to PnP configure at the next boot-time. But hardware seems to forget its configuration when the power is turned off. Some devices contain a default configuration (but not necessarily the last one used). Thus a PnP device needs to be re-configured each time the PC is powered on. Also, if a new device has been added, then it too needs to be configured too. Allocating bus-resources to this new device might involve taking some bus-resources away from an existing device and assigning the existing device alternative bus-resources that it can use instead. At present, Linux can't allocate with this sophistication (and MS Windows XP may not be able to do it either).

2.13 Starting Up the PC

When the PC is first turned on the BIOS chip runs its program to get the computer started (the first step is to check out the motherboard hardware). If the operating system is stored on the hard-drive (as it normally is) then the BIOS must know about the hard-drive. If the hard-drive is PnP then the BIOS may use PnP methods to find it. Also, in order to permit the user to manually configure the BIOS's CMOS and respond to error messages when the computer starts up, a screen (video card) and keyboard are also required. Thus the BIOS must always PnP-configure devices needed to load the operating system from the hard-drive.

Once the BIOS has identified the hard-drive, the video card, and the keyboard it is ready to start booting (loading the operating system into memory from the hard-disk). If you've told the BIOS that you have a PnP operating system (PnP OS), it should start booting the PC as above and let the operating system finish the PnP configuring. Otherwise, a PnP-BIOS will (prior to booting) likely try to do the rest of the PnP configuring of devices (but not inform the device drivers of what it did). But the drivers can still find out this by utilizing functions available in the Linux kernel.

2.14 Buses

To see what's on the PCI bus type `lspci` or `lspci -vv`. Or type `scanpci -v` for the same information in the numeric code format where the device is shown by number (such as: "device 0x122d" instead of by name, etc. In rare cases, `scanpci` will find a device that `lspci` can't find.

The boot-time messages on your display show devices which have been found on various buses (use shift-PageUp to back up thru them). See Boot-time Messages

ISA is the old bus of the old IBM-compatible PCs while PCI is a newer and faster bus from Intel. The PCI bus was designed for what is today called PnP. This makes it easy (as compared to the ISA bus) to find out how PnP bus-resources have been assigned to hardware devices.

For the ISA bus there was a real problem with implementing PnP since no one had PnP in mind when the ISA bus was designed and there are almost no I/O addresses available for PnP to use for sending configuration info to a physical device. As a result, the way PnP was shoehorned onto the ISA bus is very complicated. Whole

books have been written about it. See PnP Book. Among other things, it requires that each PnP device be assigned a temporary "handle" by the PnP program so that one may address it for PnP configuring. Assigning these "handles" is call "isolation". See ISA Isolation for the complex details.

As the ISA bus becomes extinct, PnP will be a little easier. It will then not only be easier to find out how the BIOS has configured the hardware, but there will be less conflicts since PCI can share interrupts. There will still be the need to match up device drivers with devices and also a need to configure devices that are added when the PC is up and running. The serious problem of some devices not being supported by Linux will remain.

2.15 How Linux Does PnP

Linux has had serious problems in the past in dealing with PnP but most of those problems have now been solved (as of mid 2004). Linux has gone from a non-PnP system originally, to one that can be PnP if certain options are selected when compiling the kernel. The BIOS may assign IRQs but Linux may also assign some of them or even reassign what the BIOS did. The configuration part of ACPI (Advance Configuration and Power Interface) is designed to make it easy for operating systems to do their own configuring. Linux can use ACPI if it's selected when the kernel is compiled.

In Linux, it's traditional for each device driver to do it's own low level configuring. This was difficult until Linux supplied software in the kernel that the drivers could use to make it easier on them. Today (2005) it has reached the point where the driver simply calls the kernel function: pci_enable_device() and the device gets configured by being enabled and having both an irq (if needed) and addresses assigned to the device. This assignment could be what was previously assigned by the BIOS or what the kernel had previously reserved for it when the pci or isapnp device was detected by the kernel. There's even an ACPI option for Linux to assign all devices IRQs at boot-time.

So today, in a sense, the drivers are still doing the configuring but they can do it by just telling Linux to do it (and Linux may not need to do much since it sometimes is able to use what has already been set by the BIOS or Linux). So it's really the non-device-driver part of the Linux kernel that is doing most of the configuring. Thus, it may be correct to call Linux a PnP operating system, at least for common computer architectures.

Then when a device driver finds its device, it asks to see what addresses and IRQ have been assigned (by the BIOS and/or Linux) and normally just accepts them. But if the driver wants to do so, it can try to change the addresses, using functions supplied by the kernel. But the kernel will not accept addresses that conflict with other devices or ones that the hardware can't support. When the PC starts up, you may note messages on the screen showing that some Linux device drivers have found their hardware devices and what the IRQ and address ranges are.

Thus, the kernel provides the drivers with functions (program code) that the drivers may use to find out if their device exists, how it's been configured, and functions to modify the configuration if needed. Kernel 2.2 could do this only for the PCI bus but Kernel 2.4 had this feature for both the ISA and PCI buses (provided that the appropriate PNP and PCI options have been selected when compiling the kernel). Kernel 2.6 came out with better utilization of ACPI. This by no means guarantees that all drivers will fully and correctly use these features. And legacy devices that the BIOS doesn't know about, may not get configured until you (or some configuration utility) puts its address, irq, etc. into a configuration file.

In addition, the kernel helps avoid resource conflicts by not allowing two devices that it knows about to use the same bus-resources at the same time. Originally this was only for IRQs, and DMAs but now it's for address resources as well.

If your have an old ISA bus, the program isapnp should run at boottime to find and configure pnp devices on the ISA bus. Look at the messages with "dmesg".

To see what help the kernel may provide to device drivers see the directory /usr/.../.../Documentation where one of the ... contains the word "kernel-doc" or the like. Warning: documentation here tends to be out-of-date so to get the latest info you would need to read messages on mailing lists sent by kernel developers and possibly the computer code that they write including comments. In this kernel documentation directory see pci.txt ("How to Write Linux PCI Drivers") and the file: /usr/include/linux/pci.h. Unless you are a driver guru and know C Programming, these files are written so tersely that they will not actually enable you to write a driver. But it will give you some idea of what PnP type functions are available for drivers to use.

For kernel 2.4 see isapnp.txt. For kernel 2.6, isapnp.txt is replaced by pnp.txt which is totally different than isapnp.txt and also deals with the PCI bus. Also see the O'Reilly book: Linux Device Drivers, 3rd ed., 2005. The full text is on the Internet.

2.16 Problems with Linux PnP

But there are a number of things that a real PnP operating system could handle better:

- Allocate bus-resources when they are in short supply by reallocation of resources if necessary
- Deal with choosing a driver when there is more than one driver for a physical device

Since it's each driver for itself, a driver could grab bus-resources that are needed by other devices (but not yet allocated to them by the kernel). Thus a more sophisticated PnP Linux kernel would be better, where the kernel did the allocation after all requests were in. Another alternative would be a try to reallocate resources already assigned if a devices couldn't get the resources it requested.

The "shortage of bus-resources" problem is becoming less of a problem for two reasons: One reason is that the PCI bus is replacing the ISA bus. Under PCI there is no shortage of IRQs since IRQs may be shared (even though sharing is a little less efficient). Also, PCI doesn't use DMA resources (although it does the equivalent of DMA without needing such resources).

The second reason is that more address space is available for device I/0. While the conventional I/O address space of the ISA bus was limited to 64KB, the PCI bus has 4GB of it. Since more physical devices are using main memory addresses instead of IO address space, there is still more space available, even on the ISA bus. On 32-bit PCs there is 4GB of main memory address space and much of this bus-resource is available for device IO (unless you have 4GB of main memory installed).

There was at least one early attempt to make Linux a truly PnP operating system. See http://www.astarte.free-online.co.uk. While developed around 1998 it never was put into the kernel (but probably should have been).

3. Setting up a PnP BIOS

When the computer is first turned on, the BIOS program runs before the operating system is loaded. Modern BIOSs are PnP and can configure most of the PnP devices. Some old PCI BIOSs will only configure the PCI bus. Here are some of the choices which may exist in your BIOS's CMOS menu:

- Do you have a PnP operating system?
- How are bus-resources to be controlled?

- Reset the configuration?

3.1 Do you have a PnP operating system?

Regardless of how you answer this to the BIOS, the PnP BIOS will PnP-configure the hard-drive, floppy, video card, and keyboard to make the system bootable as well as configure the LPC bus (if you have one). If you said no PnP OS then the BIOS should configure everything.

How should you answer this question to your BIOS? If you have at at least the 2.4 kernel you could answer it either way and Linux will usually work fine. Even if you have Windows 2000 or XP on the same PC, it will usually work OK either way. This is because both Windows and Linux are supposedly PnP OS's and if the OS is PnP it should be able to also handle the case where the BIOS has configured everything (if you said it wasn't PnP). But I still suggest saying that it's not a PnP OS unless there is a known reason to say otherwise.

Linux prior to the 2.4 kernel

It's not often clear whether to say yes or no. If isapnp was used by Linux, then Linux does the configuring and it was claimed that it's best to say it's a PnP OS. Why isapnp would have trouble when presented with devices already configured by the BIOS isn't clear, but such trouble sometimes happened and was fixed by stopping the BIOS from configuring (saying yes, it's a PnP OS). There were a few cases where saying no fixed a problem. So if isapnp is doing it's job OK, you should probably say it's PnP. If isapnp isn't used, no is usually best. The Linux device drivers for PCI devices should configure PCI devices OK. But for the case of PCI devices driven by non-PCI drivers, then you may say it's not PnP to get the BIOS to configure them.

Windows 2000 and XP

If you also run these Windows OS's on the same PC, you should say that you don't have a PnP OS. That's what MS suggests you do. Perhaps MS hopes that the BIOS will do a better job at configuring than Windows will. That makes sense because the BIOS should be designed for the particular idiosyncrasies of the motherboard, especially today when many devices are built into the motherboard. PnP OS = no should also be OK for Linux kernels 2.4 and higher. But for Linux kernel prior to 2.4, it's not clear which is best. (see the above subsection). So if you have problems with Linux you might try saying you have a PnP OS to satisfy Linux but this is going against what MS suggest (but will probably work OK anyway).

When the BIOS configures a device different from what Windows has in it's registry, Windows will tell you that it's finding new hardware. What it's really doing is finding old hardware that has been configured differently so it thinks it's new hardware. At any rate, it records the configuration that the BIOS has used in its registry and the device should work OK from now on.

MS Windows 95, 98 (and Me ?)

For Windows9x, MS suggest that you tell the BIOS that you have a PnP OS (the exact opposite of the case for Windows 2000 and XP). This should also be OK for Linux if you have kernel 2.4 or later. But if you have a Linux kernel prior to 2.4 then it's best for Linux to say that it's not a PnP OS. One way to resolve this dilemma is to set it up for the OS you use more frequently. Then when you boot the other OS, manually go into the BIOS and change the setting. This is a lot of bother but it's feasible if you almost never use one of the OS's. Otherwise there are better ways to resolved this dilemma.

The second way to resolve this dilemma is to get Linux to resource-configure everything. See Linux prior to

the 2.4 kernel. Then you tell the BIOS it's a PnP OS.

The third way to resolve this dilemma is to tell the BIOS it's not a PnP OS. This is going against what MS says you should do, but it's possible to get MS Windows9x to work OK if you understand what to do (and why). If you tell the BIOS it's not a PnP OS, shouldn't MS Windows detect how the BIOS has configured things and change it if it doesn't like what the BIOS has done? It should, but unfortunately, it doesn't seem to work this way.

What Windows9x seems to do when it finds hardware that is already configured by the BIOS is to just leave it alone and not reconfigure it. Now Windows9x keeps a record of the bus-resource configuration in its registry. If the BIOS configuration is different, it should either correct what's in its registry to conform to what the BIOS has set or reconfigure everything per what's in the registry. Bad news. It seems to do neither and thinks the actual configuration is the same as in the registry when in fact it's different.

But if the registry happens to contain a bus-resource configuration that is exactly the same as how the BIOS configures things, then everything will obviously work OK. A device will thus work fine if the BIOS has configured it the same as recorded in the registry. So the way to get MS Windows to work OK is to get the registry in sync with how the BIOS configures. As mentioned previously, the BIOS configures things per its ESCD (which is something like the registry for the BIOS). See The BIOS's ESCD Database. So we need to get the registry in sync with the BIOS's ESCD so that the registry and the ESCD contain the same configuration. In some cases, these two just happen to be in sync and you don't need to do anything.

One question you may think of is: how did the BIOS's ESCD and Windows registry ever get out of sync in the first place? Here's one scenario. You install Windows with the BIOS set to a PnP OS. Then Windows configures most everything and saves that configuration in its registry. Then later on you change the BIOS setting to not a PnP OS. Then upon booting, the BIOS configures everything and it doesn't do it exactly like Windows did it. Thus the actual configuration of the hardware and what Windows has in its registry are now different.

One way to try to get the Registry and the ESCD the same is to install (or reinstall) Windows when the BIOS is set for "not a PnP OS". This should present Windows with hardware configured by the BIOS. If this configuration is without conflicts, Windows will hopefully leave it alone and save it in it's Registry. Then the ESCD and the registry are in sync.

Another method is to remove devices that are causing problems in Windows by clicking on "remove" in the Device Manager. Then reboot with "Not a PnP OS" (set it in the BIOS's CMOS as you start to boot). Windows will then reinstall the devices, hopefully using the bus-resource settings as configured by the BIOS. Be warned that Windows will likely ask you to insert the Window installation CD since it sometimes can't find the driver files (and the like) even though they are still there. A workaround for this is to select "skip file" which will avoid installing the file from a CD. If the file is still on the HD, then the driver will hopefully find it OK even though the Window's install program requested you install it from a CD (which you skipped doing).

As a test I "removed" a NIC card which used a Novell compatible driver. Upon rebooting, Windows reinstalled it with Microsoft Networking instead of Novell. This meant that the Novell Client needed to be reinstalled --a lot of unnecessary work. So in a case like this it may be better to not fib to Windows95/98 but instead to get Linux to configure bus-resources.

When using a Window-Linux PC (dual boot) you might notice a change in the way the BIOS configures due to Windows9x (and other versions of Windows ??) modifying the ESCD. It supposedly does this only if you "force" a configuration or install a legacy device. See Using Windows to set ESCD. Device drivers that do

configuring may modify what the BIOS has done as will the isapnp or PCI Utilities programs if you run them.

3.2 Assigning Resources by the BIOS

Modern BIOSs allow you to manually allocate resources, primarily IRQs. There is usually an option to set the an allocation to "auto" so that the BIOS decides how to allocate the resource. "Auto" is often a good choice unless you have old legacy non-pnp ISA cards.

If you have such non-PnP cards, then it may be important to reserve resources (such as IRQ's) for these in the BIOS. Otherwise the BIOS may use these resources for some other device and create conflicts. An exception is that for some common legacy devices (such as parallel and serial ports, disk drives), the BIOS may find them (look at the screen at boot-time) so you don't need to reserve resources for them. If you've used Windows on your PC, it might be true that Windows has already told the BIOS about them by running the ICU utility (or the like) under Windows.

For PCI, the BIOS may let you assign IRQs to card slots 1, 2, 3, 4, etc. If you do this, you should know what card is in what slot. Actually, each slot has 4 PCI IRQs: A, B, C, and D. If the BIOS menu doesn't say which of these (A, B, C, D) is being assigned to an IRQ number, it's likely that it's only assigning the IRQ number to PCI IRQ A. But many PCI cards only use IRQ A so it's then just like assigning an IRQ to a slot. See PCI Interrupts

3.3 Reset the configuration?

This is a little risky to do. It will erase the BIOSs ESCD data-base of how your PnP devices should be configured as well as the list of how legacy (non-PnP) devices are configured. Never do this unless you are convinced that this data-base is wrong and needs to be remade. It was stated somewhere that you should do this only if you can't get your computer to boot. If the BIOS loses the data on legacy ISA devices, then you'll need to run ICA again under DOS/Windows to reestablish this data.

4. How to Deal with PnP Cards

4.1 Introduction to Dealing with PnP Devices

Today almost all new internal boards (cards) are Plug-and-Play (PnP). Thus, the configuring of bus-resources should, in almost all cases be entirely automatic. If a device is not working, see if it was detected, possibly by rebooting. If the device driver can't resource-configure it, then hopefully one or more of methods 2-6 will:

1. Device Driver Configures
2. /sys User Interface Configures kernel 2.6 + (not for PCI yet, other severe limitations)
3. BIOS Configures (For the PCI bus you only need a PCI BIOS, otherwise you need a PnP BIOS)
4. ISA cards only: Disable PnP by jumpers or DOS/Windows software (but many cards can't do this)
5. ISA Bus: Isapnp is a program you can always use to configure ISA PnP devices
6. PCI Utilities is for configuring the PCI bus but the device driver should handle it
7. Windows Configures and then you boot Linux from within Windows/DOS. Use as a last resort

Any of the above will set the bus-resources in the hardware but only the first one (and possibly the second) tells the driver what has been done. How the driver gets informed depends on the driver. You may need to do something to inform it. See Tell the Driver the Configuration

4.2 Device Driver Configures, Reserving Resources

Device drivers (with the help of code provided by the kernel) can be written to use PnP methods to set the bus-resources in the hardware but only for the device that they control. But many device drivers just accept what the BIOS or Linux has configured and use code provided by the kernel to find out how this device has been configured. Since the driver has checked the configuration and possibly reconfigured it, it obviously knows the configuration and there is no need for you to tell it this info. This is obviously the easiest way to do it since you don't have to do anything if the driver does it all.

If you have old pre-PnP ISA hardware, the Linux PnP software may not know about it and the bus-resources it requires. So it might erroneously allocate the resources that this old hardware needs to some other device. The result is a resource conflict but there's a way to try to avoid it. You can reserve the resources that the old ISA card needs by configuring the BIOS at boot-time (usually), the isa-pnp module or to the kernel (if the PnP is built into the kernel). For example, to reserve IRQ 5 give this argument to the isa-pnp module (or to the kernel): isapnp_reserve_irq=5. See BootPrompt-HOWTO. Instead of ..._irq there are also _io, _dma, and _mem.

For PCI devices, most drivers will configure PnP. Unfortunately, a driver could grab bus-resources that are needed by other devices (but not yet allocated to them by the kernel). Thus a more sophisticated PnP Linux kernel would be better, where the kernel did the allocation after all requests were in. See How Linux Does PnP.

4.3 /sys User Interface Configures

Starting with kernel 2.6 there's supposedly a new way for the user to resource configure using the /sys directory tree. But as of Aug. 2004, it can't be used for configuring in most cases. See The /sys Directory Tree.

4.4 BIOS Configures

Intro to Using the BIOS to Configure PnP

If you have a PnP BIOS, it can configure the hardware. If the driver can't do it, the BIOS probably can. This means that your BIOS reads the resource requirements of all devices and configures them (allocates bus-resources to them). It is a substitute for a PnP OS except that the BIOS doesn't match up the drivers with their devices nor tell the drivers how it has done the configuring. It should normally use the configuration it has stored in its non-volatile memory (ESCD). If it finds a new device or if there's a conflict, the BIOS should make the necessary changes to the configuration and may not use the same configuration as was in the ESCD. In this case it should update the ESCD to reflect the new situation.

Your BIOS needs to support such configuring and there have been cases where it doesn't do it correctly or completely. The BIOS may need to be told via the CMOS menu that it's not a PnP OS. While many device drivers will be able to automatically detect what the BIOS has done, in some cases you may need to determine it (not always easy). See What Is My Current Configuration? A possible advantage to letting the BIOS do it is that it does its work before Linux starts so it all gets done early in the boot process.

Most BIOS made after about 1996 ?? can resource-configure both the PCI and ISA buses. But it's been claimed that some older BIOSs can only do the PCI. And of course, for PCs with only the PCI bus, the BIOS only needs to do PCI. To try to find out more about your BIOS, look on the Web. Please don't ask me as I don't have data on this. The details of the BIOS that you would like to know about may be hard to find (or not

available). Some old BIOS's may have minimal PnP capabilities and seemingly expect the operating system to do it right. If this happens you'll either have to find another method or try to set up the ESCD database if the BIOS has one. See the next section.

The BIOS's ESCD Database

The BIOS maintains a non-volatile database containing a PnP-configuration that it will try to use (if you claim that it's not a PnP OS). It's called the ESCD (Extended System Configuration Data). Again, the provision of ESCD is optional but most PnP-BIOSs have it. The ESCD not only stores the resource-configuration of PnP devices but also stores configuration information of non-PnP devices (and marks them as such) so as to avoid conflicts. The ESCD data is usually saved on a chip and remains intact when the power is off, but sometimes it's kept on a hard-drive??

The ESCD is intended to hold the last used configuration. But since Linux can change how devices are configured (including the user using isapnp or pci utilities) then the ESCD will not know about this and will not save this configuration in the ESCD. A good PnP OS might update the ESCD so you can use it later on for a non-PnP OS (like standard Linux). MS Windows9x does this only in special cases. See Using Windows to set ESCD. Starting with kernel 2.6, Linux is capable of modifying the ESCD but it's not used yet (as of Aug. 2004).

To use what's set in ESCD be sure you've set "Not a PnP OS" or the like in the BIOS's CMOS. Then each time the BIOS starts up (before the Linux OS is loaded) it should configure things this way. If the BIOS detects a new PnP card which is not in the ESCD, then it must allocate bus-resources to the card and update the ESCD. It may even have to change the bus-resources assigned to existing PnP cards and modify the ESCD accordingly.

There's a program that you may use to view the contents of the ESCD. It shows IRQs and IO addresses etc., but device names are missing (only EISA device-ID numbers). It's at: Index of /home/gunther.mayer/lsescd

If each device saved its last configuration in its hardware, hardware configuring wouldn't be needed each time you start your PC. But it doesn't work this way. So all the ESCD data needs to be kept correct if you use the BIOS for PnP. There are some BIOSs that don't have an ESCD but do have some non-volatile memory to store info regarding which bus-resources have been reserved for use by non-PnP cards. Many BIOSs have both.

Using Windows to set the ESCD

Eventually the Linux kernel may set the ESCD. Starting with kernel 2.6, a function in the new code could do it provided the kernel has been compiled with PNPBIOS. But it currently sits in the code unused.

If the BIOS doesn't set up the ESCD the way you want it (or the way it should be) then it would be nice to have a Linux utility to set the ESCD. One may resort to attempting to use Windows for this (if you have it on the same PC) to do this.

There are three ways to use Windows to try to set/modify the ESCD. One way is to use the ICU utility designed for DOS or Windows 3.x. It should also work OK for Windows 9x/2k ?? Another way is to set up devices manually ("forced") under Windows 9x/2k so that Windows will put this info into the ESCD when Windows is shut down normally. The third way is only for legacy devices that are not plug-and-play. If Windows knows about them and what bus-resources they use, then Windows should put this info into the ESCD.

If PnP devices are configured automatically by Windows without the user "forcing" it to change settings, then such settings probably will not make it into the ESCD. Of course Windows may well decide on its own to configure the same as what is set in the ESCD so they could wind up being the same by coincidence.

Windows 9x are PnP operating systems and automatically PnP-configure devices. They maintain their own PnP-database deep down in the Registry (stored in binary Windows files). There is also a lot of other configuration stuff in the Registry besides PnP-bus-resources. There is both a current PnP resource configuration in memory and another (perhaps about the same) stored on the hard disk. To look at this in Windows98 or to force changes to it you use the Device Manager.

In Windows98 there are 2 ways to get to the Device Manager: 1. My Computer --> Control Panel --> System Properties --> Device Manager. 2. (right-click) My Computer --> Properties --> Device Manager. Then in Device Manager you select a device (sometimes a multi-step process if there are a few devices of the same class). Then click on "Properties" and then on "Resources". To attempt to change the resource configuration manually, uncheck "Use automatic settings" and then click on "Change Settings". Now try to change the setting, but it may not let you change it. If it does let you, you have "forced" a change. A message should inform you that it's being forced. If you want to keep the existing setting shown by Windows but make it "forced" then you will have to force a change to something else and then force it back to its original setting.

To see what has been "forced" under Windows98 look at the "forced hardware" list: Start --> Programs --> Accessories --> System Tools --> System Information --> Hardware Resources --> Forced Hardware. When you "force" a change of bus-resources in Windows, it should put your change into the ESCD (provided you exit Windows normally). From the "System Information" window you may also inspect how IRQs and IO ports have been allocated under Windows.

Even if Windows shows no conflict of bus-resources, there may be a conflict under Linux. That's because Windows may assign bus-resources differently than the ESCD does. In the rare case where all devices under Windows are either legacy devices or have been "forced", then Windows and the ESCD configurations should be identical.

Adding a New Device (under Linux or Windows)

If you add a new PnP device and have the BIOS set to "not a PnP OS", then the BIOS should automatically configure it and store the configuration in ESCD. If it's a non-PnP legacy device (or one made that way by jumpers, etc.) then here are a few options to handle it:

You may be able to tell the BIOS directly (via the CMOS setup menus) that certain bus-resources it uses (such as IRQs) are reserved and are not to be allocated by PnP. This does not put this info into the ESCD. But there may be a BIOS menu selection as to whether or not to have these CMOS choices override what may be in the ESCD in case of conflict. Another method is to run ICU under DOS/Windows. Still another is to install it manually under Windows 9x/2k and then make sure its configuration is "forced" (see the previous section). If it's "forced" Windows should update the ESCD when you shut down the PC.

4.5 ISA cards only: Disable PnP ?

PCI devices are inherently PnP so it can't be disabled. But a few ISA devices once had options for disabling PnP by jumpers or by running a Windows program that comes with the device (jumperless configuration). If the device driver can't configure it, this will avoid the possibly complicated task of doing PnP configuring. Don't forget to tell the BIOS that these bus-resources are reserved. But since Linux support for PnP has improved, you usually don't want to disable PnP. Here's some more arguments in favor of PnP:

1. If you have MS Windows on the same machine, then you may want to allow PnP to configure devices differently under Windows from what it does under Linux.
2. The range of selection for IRQ numbers (or port addresses) etc. may be too limited unless you use PnP.
3. You might have a Linux device driver that uses PnP methods to search for the device it controls.
4. If you need to change the configuration in the future, it may be easier to do this if it's PnP (no setting of jumpers or running a Dos/Windows program).

Once configured as non-PnP devices, they can't be configured by PnP software or a PnP-BIOS (until you move jumpers and/or use the Dos/Windows configuration software again).

4.6 ISA Bus: Isapnp (part of isapnptools)

The `isapnp` standalone program is only for PnP devices on the ISA bus (non-PCI). It was much needed prior to the 2.4 kernels. After the 2.4 kernel, which provided functionality to allow drivers deal with ISA PnP, the isapnp standalone program is less significant. Also, the BIOS may configure ISA PnP satisfactory. But the isa-pnp module (or the equivalent built into the kernel) is now very significant since various ISA device drivers call on it to configure bus-resources. Prior to kernel 2.6 it resulted a /proc/isapnp "file" which may be used to manually configure (see isapnp.txt in the kernel documentation).

In some cases Linux distributions have been set up to run isapnp automatically at startup. It's still done in 2004 but it isn't really needed if the device drivers work well. If you need to set it up yourself much of the documentation for isapnp is difficult to understand unless you know the basics of PnP. This HOWTO should help you understand it as well the FAQ that comes with isapnp. Running the Linux program "isapnp" at boot-time will configure such devices to the resource values specified in /etc/isapnp.conf. Its possible to create this configuration file automatically but you then should edit it manually to choose between various options. Then to let the driver know the resources, you often need to specify them as parameters to the appropriate modules (drivers). This is done with configuration files, often in the /etc directory. Look there for files named mod*, etc. If the driver is built into the kernel, then they may sometimes be given as a parameter to the kernel. See BootPrompt-HOWTO.

With isapnp there once was a problem where a device driver which is built into the kernel may run too early before isapnp has set the address, etc. in the hardware. This resulted in the device driver not being able to find the device. The driver tries the right address but the address hasn't been set yet in the hardware. Is this still a problem ??

If your Linux distribution automatically installed isapnptools, isapnp may already be running at startup. In this case, all you need to do is to edit /etc/isapnp.conf per "`man isapnp.conf`". Note that this is like manually configuring PnP since you make the decisions as to how to configure as you edit the configuration file.

If the configuration file is wrong or doesn't exist, you can use the program "pnpdump" to help create the configuration file. It almost creates a configuration file for you but you must skillfully edit it a little before using it. It contains some comments to help you edit it. While the BIOS may also configure the ISA devices (if you've told it that you don't have a PnP OS), isapnp will redo it.

The terminology used in the /etc/isapnp.conf file may seem odd at first. For example for an IO address of 0x3e8 you might see "(IO 0 (BASE 0x3e8))" instead. The "IO 0" means this is the first (0th) IO address-range that this device uses. Another way to express all this would be: "IO[0] = 0x3e8" but isapnp doesn't do it this way. "IO 1" would mean that this is the second IO address range used by this device, etc. "INT 0" has a similar meaning but for IRQs (interrupts). A single card may contain several physical devices but the above

explanation was for just one of these devices.

4.7 PCI Utilities

The package PCI Utilities (= pciutils, sometimes called "pcitools"), allows one to manually PnP-configure the PCI bus (with difficulty). "lspci" or "scanpci" lists bus-resources while "setpci" sets resource allocations (except IRQs) in the hardware devices. It appears that setpci is mainly intended for use in scripts and one needs to understand the details of the PCI configuration registers in order to use it. That's a topic not explained here nor in the manual page for setpci.

People have used this to configure PCI devices where the driver failed to do it. An example is found in my Modem-HOWTO and Serial-HOWTO in the subsection "PCI: Enabling a disabled port". However, enabling a device is of no use unless you have a working driver for the device.

4.8 Windows Configures

This method uses MS Windows to configure and should be used only if all else fails. If you have Windows9x (or 2k) on the same PC, then just start Windows and let it configure PnP. Then start Linux from Windows (or DOS) using, for example, loadlin.exe. But there may be a problem with IRQs for PCI devices. As Windows shuts down (without any messages) to make way for Linux, it may erase (zero) the IRQ which is stored in one of the PCI device's configuration registers. Linux will complain that it has found an IRQ of zero.

The above is reported to happen if you start Linux using a shortcut (PIF file). But a workaround is reported where you still use the shortcut PIF. A shortcut is something like a symbolic link in Linux but it's more than that since it may be "configured". To start Linux from DOS you create a batch file (script) which starts Linux. (The program that starts Linux is in the package called "loadlin"). Then create a PIF shortcut to that batch file and get to the "Properties" dialog box for the shortcut. Select "Advanced" and then check "MS-DOS mode" to get it to start in genuine MS-DOS.

Now here's the trick to prevent zeroing the PCI IRQs. Click "Specify a new MS-DOS configuration". Then either accept the default configuration presented to you or click on "Configuration" to change it. Now when you start Linux by clicking on the shortcut, new configuration files (Config.sys and Autoexec.bat) will be created per your new configuration.

The old files are stored as "Config.wos and Autoexec.wos". After you are done using Linux and shut down your PC then you'll need these files again so that you can run DOS the next time you start your PC. You need to ensure that the names get restored to *.sys and *.bat. When you leave Windows/DOS to enter Linux, Windows is expecting that when you are done using Linux you will return to Windows so that Windows can automatically restore these files to their original names. But this doesn't happen since when you exit Linux you shut down your PC and don't get back to Windows. So how do you get these files renamed? It's easy, just put commands into your "start-Linux" batch file to rename these files to their *.bat and *.sys names. Put these renaming commands into your batch file just before the line that loads Linux.

Also it's reported that you should click on the "General" tab (of the "Properties" dialog of your shortcut) and check "Read-only". Otherwise Windows may reset the "Advanced Settings" to "Use current MS-DOS configuration" and PCI IRQs get zeroed. Thus Windows erases the IRQs when you use the current MS-DOS configuration but doesn't erase when you use a new configuration (which may actually configure things identical to the old configuration). Windows does not seem to be very consistent.

4.9 PnP Software/Documents

- Isapnptools homepage
- Proposal for a Configuration Manager for Linux 1999 (Never got into kernel but Linux is slowly "evolving" in this direction).
- PnP Specs. from Microsoft
- Book: PCI System Architecture, 4th ed. by Tom Shanley +, MindShare 1999. Covers PnP-like features on the PCI bus.
- Book: Plug and Play System Architecture, by Tom Shanley, Mind Share 1999. Details of PnP on the ISA bus. Only a terse overview of PnP on the PCI bus.
- Book: Programming Plug and Play, by James Kelsey, Sams 1995. Details of programming to communicate with a PnP BIOS. Covers ISA, PCI, and PCMCIA buses.

5. Tell the Driver the Configuration ??

5.1 Introduction

A modern driver for a device will find out the bus-resource configuration without you having to tell it anything. It may even set the bus-resources in the hardware using PnP methods. Some drivers have more than one way to find out how their physical device is configured. In the worst case you must hard-code the bus-resources into the kernel (or a module) and recompile.

In the middle are cases such as where you run a program to give the bus-resource info to the driver or put the info in a configuration file. In some cases the driver may probe for the device at addresses where it suspects the device resides (but it will never find a PnP device if it hasn't been enabled by PnP methods). It may then try to test various IRQs to see which one works. It may or may not automatically do this.

In the modern case the driver should use PnP methods to find the device and how the bus-resources have been set by the BIOS, etc. but will not actually set them. It may also look at some of the "files" in the /proc directory.

One may need to "manually" tell a driver what bus-resources it should use. You give such bus-resources as a parameter to the kernel or to a loadable module. If the driver is built into the kernel, you pass the parameters to the kernel via the "boot-prompt". See The Boot-Prompt-HOWTO which describes some of the bus-resource and other parameters. Once you know what parameters to give to the kernel, one may put them into a boot loader configuration file. For example, put append="...". into the lilo.conf file and then use the lilo command to get this info into the lilo kernel loader.

If the driver is loaded as a module, in many cases the module will find the bus-resources needed and then set them in the device. In other cases (mostly for older PCs) you may need to give bus-resources as parameters to the module. Parameters to a module (including ones that automatically load) may be specified in /etc/modules.conf. There are usually tools used to modify this file which are distribution-dependent. Comments in this file should help regarding how to modify it. Also, any module your put in /etc/modules will get loaded along with its parameters.

While there is much non-uniformity about how drivers find out about bus-resources, the end goal is the same. If you're having problems with a driver you may need to look at the driver documentation (check the kernel documentation tree). Some brief examples of a few drivers is presented in the following sections:

5.2 Serial Port Driver Example

For PCI serial ports (and for ISA PnP serial ports after 2.4 kernels) the serial driver detects the type of serial port and PnP configures it. Unfortunately, there may be some PCI serial ports that are not supported yet.

For the standard ISA serial port with very old older versions of the kernel and serial driver (not for multiport cards) the driver probes two standard addresses for serial ports. It doesn't probe for IRQs but it just assigns the "standard" IRQ to the first two serial ports. This could be wrong.

For anything else the configuration file for the `setserial` program must be manually modified. See Serial-HOWTO for more details. You use setserial to inform the driver of the IO address and Setserial is often run from a start-up file. In newer versions there is a /etc/serial.conf file (or /var/lib/setserial/autoconfig that you "edit" by simply using the setserial command in the normal way and what you set using `setserial` is saved in the `serial.conf` configuration file. The `serial.conf` file should be consulted when the `setserial` command runs from a start-up file. Your distribution may or may not set this up for you.

There are two different ways to use `setserial` depending on the options you give it. One use is used to manually tell the driver the configuration. The other use is to probe at a given address and report if a serial port exists there. It can also probe this address and try to detect what IRQ is used for this port.

Even with modern kernels, setserial is sometimes needed if the driver fails to detect the serial port, or if you have very old hardware.

6. How Do I Find Devices and How Are They Configured?

6.1 Finding and How-Configured Are Related

Once you find your hardware, the same program that found it usually tells you how it's configured. So finding out how it's configured is usually the same procedure as finding the hardware.

6.2 Devices May Have Two "Configurations"

Here "configuration" means the assignment of PnP bus-resources (addresses, IRQs, and DMAs). For each device, there are two parts to the configuration question:

1. What does the driver think the hardware configuration is?
2. What configuration (if any) is actually set in the device hardware?

Each part should have the same answer (the same configuration). The configuration of the device hardware and its driver should obviously be the same (and usually is). But if things are not working right, it could be because there's a difference. This means that the driver has incorrect information about the actual configuration of the hardware. This spells trouble. If the software you use doesn't adequately tell you what's wrong (or automatically configure it correctly) then you need to investigate how your hardware devices and their drivers are configured. While Linux device drivers should "tell all", in some cases it may not be easy to determine what has been set in the hardware.

Another problem is that when you view configuration messages on the screen you need to know whether the reported configuration is that of the device driver, the device hardware, or both. If the device driver has either set the configuration in the hardware or has otherwise checked the hardware then the driver should have the

correct information.

But sometimes the driver has been provided incorrect resources by a script, configuration file, by incorrect resource parameters given to a module, or perhaps just hasn't been told what the resources are and tries to use incorrect default resources. For example, one can uses "setserial" to tell the serial port driver an incorrect resource configuration and the driver accepts it without question. But the serial port doesn't work right (if at all).

6.3 Finding Hardware

A common problem is that the software doesn't detect your device and/or determine the right driver for it. For PnP devices, detecting them is easy via PnP software except for the unusual case where the hardware has been disabled. The BIOS can sometimes be set to disable PnP devices or a jumper/switch on the physical device itself could disable it. In such a cases, the hardware can't be detected at all until you either reconfigure the BIOS or change a jumper/switch.

Since the PCI bus is inherently PnP, there are no hidden devices. Even though PnP devices are easy to find by PnP methods, if the driver doesn't use PnP methods but uses the old method of probing for them at likely address, they may not be found. This is because that, until the resources are set in a PnP device (by the BIOS or Linux), the device may have no address at all, so probing at likely address yields nothing. For the old ISA bus, some of the devices may be non-PnP and thus the old probing methods may find them. So many drivers still probe at likely address, in addition to using PnP methods (= PnP probing which is sometimes also just called "probing").

Ways to Find Hardware Devices (and their configurations): (follow link to more details)

- Check the BIOS to make sure they are not disabled
- Watch the Boot-time Messages on the screen
- Look in The /proc Directory Tree
- Tools for Detecting and/or Configuring all Hardware lsdev, hwinfo, discover, kudzu
- Tools for Detecting and/or Configuring One Type of Hardware
- PCI: PCI Bus Inspection
- ISA Bus: ISA Bus Introduction
- ISA Bus: PnP cards
- ISA Bus: For Non-PnP Cards
- ISA Bus: For Cards with jumpers
- ISA Bus: If Neither PnP nor jumpers
- Use MS Windows

6.4 Boot-time Messages

Significant info on the configuration may be obtained by reading the messages from the BIOS and from Linux that appear on the screen when you first start the computer. These messages often flash by too fast to read but once they stop type Shift-PageUp a few times to scroll back to them. To scroll forward thru them type Shift-PageDown. Typing "dmesg" at any time to the shell prompt will show only the Linux kernel messages and may miss some of the most important ones (including ones from the BIOS). The messages from Linux may sometimes only show what the device driver thinks the configuration is, perhaps as told it via an incorrect configuration file. Checking log files in /var/log may also be useful.

For the PCI bus, the notation: 00:1a:0 means the PCI bus 00 (the main PCI bus), PCI card (or chip) 1a, and function 0 (the first device) on the card or chip. The 2nd device on the card (or chip) 08 would be: 00:08:1.

The BIOS messages display first and will show the actual hardware configuration at that time, but isapnp, or pci utilities, or device drivers may change it later. In some cases it doesn't show devices that the BIOS didn't configure.

If the BIOS messages don't show as you back up to the start of the BIOS messages using Shift-PageUp, try freezing them as they flash by, by hitting the "Pause" key as soon as the first words flash on the screen. Press any key to resume. It's often tricky to hit Pause exactly at the right time. Be sure to hold down the "Shift" key before hitting "Pause" since "Pause" is a shifted key. If you miss, hit Ctrl-Alt-Del when Linux starts booting to reboot and try again. Once the messages from Linux start to appear, it's too late to use "Pause" since it will not freeze the messages from Linux.

To set things in the BIOS such as IRQs reserved for legacy hardware, serial port addresses, etc. you need to get into the BIOS (CMOS) setup menus at boot time. Each BIOS brand has different keys you need to hold down to do this. There are lists on the Internet. Sometimes by freezing the BIOS messages or watching the screen, the key you need to press will be indicated in a message such as "Press DEL for setup". But it may flash by so fast that you miss it. Of course, you don't set stuff in the BIOS that you don't understand, or your PC may become disabled.

Messages from the BIOS at boot-time tell you how the hardware configuration was then. The current configuration may still be the same since Linux should hopefully accept what the BIOS has done if it's OK. Messages from Linux may be from drivers that used kernel PnP functions to inspect and/or set bus-resources. These should be correct, but beware of messages that only show what the driver was told from a configuration file. It could be wrong. Of course, if the device works fine, then it's likely configured the same as the driver.

6.5 The /proc Tree

Starting with Kernel 2.6, in addition to the /proc directory tree, there's also a /sys tree See The /sys Tree. These trees are useful for finding resource configurations and devices. The "files" in them represent data in the kernel memory and don't exist at all on you harddrive. Programs such as lspci get their info from the /proc tree so such programs should display the results in more readable form than directly inspecting the "files" in /proc. Here are 4 /proc "files" that show resources which have been registered in the kernel by device drivers.

Since Linux's plug-and-play works by letting device drivers allocate resources for their device, there may be no listing of resources used by some of your hardware if the driver hasn't yet requested that such resources be reserved. For the case of kernel modules (loadable device drivers), if the module hasn't loaded yet, the kernel doesn't know about any resources it needs. Sometimes, the module only loads when you start an application that needs it. So if certain hardware is missing from these "files" in /proc, it may mean that the hardware hasn't yet been used. For example, even though your floppy drive has a floppy disk in it and is ready to use, the interrupt for it will not show up unless its in use.

/pts shows I/O addresses. If there's a mistake (wrong address) it means trouble since the device will not get bytes sent to it.
/proc/iomem shows registered IO memory addresses.
/proc/interrupts shows the interrupts currently in use.
/proc/dma shows the dma (Direct Memory Access) ISA dma channel allocations.

In the past, the author observed the listing of interrupts that didn't exist. In some cases it showed that a few such interrupts were actually sent. This could be due to the issuing of erroneous interrupts due to hardware defects.

/proc/bus/ has subdirectories (subfolders) input/, pci/, and isapnp/. The format of most of the files in this directory is very cryptic, often just a copy of the bytes in the configuration space. So, use them only as a last resort. The input/ subdirectory has information on input devices such as the keyboard and mouse. It's not as cryptic as the other directories under /proc/bus/ and might yield some useful information about input devices that are PS2 or on the LPC bus (See LPC Bus). Unfortunately, what I've seen doesn't say that it's on the LPC bus when it likely is. In /pci/00/ there is one binary file for each pci device where the file names are the pci-slot-numbers (also called pci-slot-names). The 00 means pci bus 0.

6.6 The /sys Tree

Starting with kernel 2.6 there's a new /sys directory for PnP configuration. It's a sysfs type of file system and it's something like the /proc filesystem since the "files" represent information in the kernel memory and are not on your harddrive. But it's not as useful as the /proc filesystem. Originally (in the 2.5 kernels) it was called "driver file system" of type "driverfs".

In the sysfs, each device which exists on your system has it's own directory which contains files showing the resources allocated to it. Such device directories have names like 0000:00:12.0@ or 00:06@. What devices are these? The first is a PCI card in "slot" 12 of your PC. The slot may actually be labeled PCI2 inside your PC (2 instead of 12). That's because low numbered "slots" are used for built-in devices on the motherboard that don't use any physical slots. In this example, "slots" 1-10 would be built-in and actual slots 11-14 are labeled 1-4. By typing "lspci" you'll be able to match the numbers (like 0000:00:12.0) to names (like IDE interface). Type "lspci -v" or "lspci -vv" to see more.

Well then, what is 00:06 ? It's an ISA card (or built-in device) but it's not ISA slot 6 (like the PCI numbering). When a search was made for ISA-PNP devices, it was the 6th one found. More precisely, it was the 7th one found since there's a device numbered: 00:00. So how does one identify them? Well, you could type: "cat */*" and display all the files for all the devices, but even then you don't see the device names (but do see info from which you can identify them). This inconvenience will hopefully be fixed in the future.

Not only do these files supply information on the bus-resource configuration (in somewhat cryptic format) and drivers (in "driver" directories), but in the future, you should be able to use them to change the resource configuration. Right now (Aug 2004) you can't configure the PCI bus with it. A serious limitation is that per the present "driver model" you can't change the resource of a device that has been assigned to a driver which likely means that you'll need to unload the driver module in order to use it. If the driver is built in, there's no hope. These serious limitations will hopefully be eliminated in the future. In the kernel documentation is a file: "pnp.txt" telling how to configure. As of Aug. 2004, it was much out-of-date but the author is working on an update. Using the /sys tree to configure resources is known as the "Linux Plug and Play User Interface".

The other part of "Linux Plug and Play" is the kernel interface used by device drivers. This has changed a lot starting with kernel 2.6 but most drivers are still using the old interface (as of Aug. 2004). It's possible also for drivers (or you) to use the "user interface" which needs improvement.

6.7 PCI Bus Inspection

It's easy to find out what bus-resources have been assigned to devices on the PCI bus with the "lspci" and/or "scanpci" commands The options -v or -vv will show more detail. In some cases, "scanpci" will find a device

that "lspci" can't find. That's because "scanpci" directly searches for devices on the pci bus (via the configuration space) and doesn't use data obtained by the kernel (where it could be wrong due to a kernel bug --I've just found such a case).

This info in more cryptic format is found in "files" located in the `/sys` and `/proc` trees. In `/sys/bus/pci/devices` the file `vendor` will contain the vendor id number such as 0x4B8C, etc. In still more cryptic format it's in `/proc/bus/pci`. Such information in older kernels prior to kernel 2.6, was in `/proc/pci` (non-cryptic but IRQs in hexadecimal) or in `/proc/buspci/devices` (cryptic display).

In most cases for PCI you will only see how the hardware is now configured and not what resources are required. In some cases you only see the base addresses (the starting addresses of the range) but not the ending addresses. If you see the entire range then you can determine how many bytes of address resources are needed.

6.8 ISA Bus Introduction

For cards on the ISA bus, it's not as simple as for the PCI bus which is inherently PnP. Later ISA cards were PnP but older ones were not. Also, some cards that are PnP had their PnP disabled by special software which runs only on MS. The non PnP cards are configured by jumpers on the card or by MS software.

6.9 ISA PnP cards

If it's a PnP card you may try running `pnpdump --dumpregs` but it's not a sure thing. The results may be seem cryptic but they can be deciphered. Don't confuse the read-port address which `pnpdump` uses for communication with PnP cards with the I/O address of the found device. They are not the same.

6.10 LPC Bus

LPC (Low Pin Count) is a bus-like interface often used on laptops and increasingly used on desktops too. To find out if you have LPC type "lspci" and look for "LPC". There are other words next to "LPC" such as "ISA Bridge ... LPC Interface Controller" or "LPC Bridge", etc. LPC is not really ISA but it substitutes for an ISA bus.

The old ISA bus was slow and devices that needed more speed were put on the newer PCI but. But devices that didn't need high speed were often implemented by chips on the motherboard and remained on the ISA bus even though there were no slots for any ISA cards. Then the LPC bus came along to replace what remained of the ISA bus. LPC is much smaller than ISA and just as fast since it runs at 4 times the clock speed of ISA. Its multiplexed bus for data/address and control is only 4 bits wide. To send a byte requires splitting the byte into 2 half-bytes and then putting them back together. So its clear why it's "Low Pin Count" = LPC. There's also a few other lines in the bus.

This small LPC interface is used for slow "legacy" devices such as serial ports, parallel ports, and floppy drives. So a computer using LPC will have all fast devices on the PCI bus, etc. and slow (legacy) devices on the LPC bus interface. All LPC devices will be on-board; there are no LPC slots.

LPC has no standards for Plug-and-Play configuring but says that the BIOS or ACPI should do the configuring. Devices on this bus sometimes use isapnp. Linux support for LPC as of late 2004 was very much incomplete but Linux has some support for the configuring aspects of ACPI. Sometimes a BIOS menu lets one manually PnP-configure devices on the LPC bus but it may not tell you that the device resides on LPC.

A major chip on the LPC bus is the superio chip which contains legacy IO devices: serial and parallel ports, floppy controller, keyboard controller, mice, etc. BIOS data may also reside on the LPC bus. The keyboard and mouse (input devices) should be listed in /proc/bus/input/devices but instead of seeing "lpc" it seems to show "isa0060/serio0, etc. even though it's on the lpc bus and not the isa bus.

6.11 X-bus

Before the LPC bus became popular, there was an "X-bus" (not covered in this HOWTO) which served the same purpose as the LPC bus but wasn't so compact as LPC. Some PCs have both LPC and an X-bus.

6.12 Non-PnP Cards

In contrast to PnP cards, non-PnP cards always have their resources set in the hardware. That is they always have an address and IRQ unless there is a jumper setting, etc. for disabling the device. Sometimes the resources used can be found by probing done by the device driver or by other software that does probing. For example "scanport" (Debian only ??) probes most IO port address and may find ISA devices. But be warned that it might hang your PC. Sometimes it will fail to find hardware that's actually there (since the hardware has the default 0xff in it's registers). Even if It finds the hardware it will not show the IRQ nor will it positively identify the hardware.

So one way to try to find such hardware is to start a driver, which may probe for such hardware. By looking at the boot-time messages, you might see a driver start and find the hardware. Otherwise, you may need to find a driver and start it (for example, by having it load as a module).

Finding the right driver may be difficult. Sometimes there just isn't any driver since some devices aren't (yet ?) supported by Linux. To determine which driver you need, look at any documentation which might identify the card. If this fails, look on the card itself, including important names/numbers on the chips. But the identification of the driver module you need may not be anywhere on the card. You could find the FCC id on the card and then search the Internet with the FCC id number to try to find more information about the card (or the chips on it).

6.13 Non-PnP Cards with jumpers

If the card has jumpers to set the resources (configuration) then one may look at how the jumpers are set. There are some cards that had both PnP and jumpers. They worked like jumper cards if PnP was somehow disabled. Sometimes a card has labels on it showing how to set the jumpers (or at least gives some clue). You may need the documentation that came with the card (either printed or on a floppy disk). Perhaps you can find it on the Internet.

6.14 Neither PnP nor jumpers

One the most difficult cases is where software running under MS has been used to configure either a non-PnP card or a PnP card where PnP has been disabled by the same MS software. So you can't configure it by PnP nor by jumpers. In this case your only hope is to probe for addresses as described in Non-PnP Cards. Or try to find the MS software that configured it.

6.15 Tools for Detecting and/or Configuring all Hardware

In a duplication of effort, various major distributions of Linux developed their own tools for detection and/or configuration of hardware. This configuring is usually a lot more than just the resource type configuring of Plug-and-Play. It's configuring in general which is mostly beyond the scope of this howto.

Then other distributions, such as Debian, might obtain copies of the tool and offer it to their users as an option, or as a troubleshooting tool. These tools likely make use of the standard Linux tools for detecting hardware such as "lspci". In the following list of tools, the name of the distribution that developed it is in parentheses, but the tool is likely available also in other distributions.

- hardinfo
- hwinfo (SuSE) detects move stuff than discover
- discover (Progeny, used by Debian)
- Kudzu (RedHat) detects and configures
- lsdev (standard Linux command)
- hwsetup-knoppix (Knoppix, based on Kudzu)

6.16 Tools for Detecting and Configuring One Type of Hardware

There are various tools available to find and possibly configure various type of devices. This configuring is configuring in general which is not covered by this howto.

- read-edid (get-edid): gets parameters of VESA monitors (except very old ones)
- sndconfig: for soundcards
- printtool: printers, must have X-window running
- pconf-detect: parallel ports
- gpm-mouse-test:detects and tests mice
- mdetect: detects and configures mice Does it know about the mice devices in /dev/input/?
- nictools-pci (and nictools-nopci) for ethernet cards
- hdparm: configure hard drive hardware
- hotplug: used by kernel
- xvidtune: tune video for use with Xwindows (See XFree86-Video-Timings-HOWTO)

6.17 Use MS Windows

Some people have attempted to use Windows to see how bus-resources have been set up. Unfortunately, since PnP hardware forgets its bus-resource configuration when powered down, the configuration may not be the same under Linux. For non PnP hardware (or where someone has disabled PnP inside the device by jumpers or Windows software), then using Windows should work OK. Even for PnP, it often turns out to be the same because in many cases both Windows and Linux simply accept what the BIOS has set. But where Windows and/or Linux do the configuring, they may do it differently. So don't count on PnP devices being configured the same.

7. PCI Interrupts

7.1 Introduction

Each PCI device that needs an interrupt comes with a fixed PCI interrupt that can't be changed. It's designated by a slot number and a letter A, B, C, or D. Example 3:B. But this PCI interrupt is mapped (routed or redirected) to an interrupt number like say 21 by a chip on the motherboard.

This routing is done by a "programmable interrupt router" = PIR. Alternatively, an interrupt line may just routed directly (without any PIR). If there's a PIR (router) it can be programmed by the BIOS or by Linux. Thus a PCI device's interrupt may be sometimes be changed, not by sending the interrupt on a different wire but by changing the routing of the pulse on that wire by programming the PIR. When the routing changes, the interrupt provide by this new routing is written in a configuration register located in the device chip.

7.2 History: From ISA to PCI Interrupts

Before the PCI bus, PCs used the ISA bus and then during the transition to the newer PCI bus, most PC computers used both the PCI and ISA busses. The ISA bus had all interrupt lines going to every card so any card could change its irq number just by sending out its interrupt signal on a different line (on a different pin). All the interrupt signals were sent to the in interrupt controller which then signalled the CPU to temporarily stop whatever it was doing and run driver code to service the interrupt.

When PCI first appeared, the simple solution was just to map the PCI interrupts to available ISA interrupts that weren't being used. This required the use of "programmable interrupt router" = PIR (hardware) to do this mapping. But since there were only 15 such interrupts, it was common to put many PCI devices on just the few available interrupts. To solve this problem is simple: provide new hardware to increase the number of interrupts. The result was the APIC. But it was slow to be adopted since the ability of the PCI bus to share interrupts eased the interrupt shortage problem. So APIC was mostly used where it was needed for dual processors.

7.3 Advanced Programmable Interrupt Controller (APIC)

This can provide (depending on the model) 16, 24, 32, or 64 interrupts, etc. It also can handle the routing of interrupts from one CPU to another for cases of multiple CPUs. See the file "IO-APIC" in the i386 directory of the kernel documentation and the ACPI-HOWTO. Don't confuse APIC with ACPI (Advanced Configuration and Power Interface) which may be used by the kernel to configure the APIC.

The actual APIC controller that is connected to the interrupt lines is an I/O APIC (or IO-APIC or IOAPIC). By using more than one IO-APIC one may obtain more interrupts and they are numbered so as to be unique. For example, the first controller could have input pins 0-23 and the second would call its input pins 24-47, resulting in 48 interrupts numbered 0-47. But a few people find they have high interrupt numbers. Could it be that the second IO-APIC is starting with a higher base number than it should, leaving a long gap of non-existent irqs?

Besides IO-APICs there are local APICs (LAPIC) which are part of each CPU. The IO-APIC does it work by communicating with the LAPICs inside the CPUs.

When APIC was introduced, the old ISA PICs were also retained giving one a choice of whether or not to use APIC or ISA's PIC (which is sometimes just called PIC or XT-PIC in */proc/interrupts*; the "XT" comes from IBM's XT PC which was IBM's second model PC in 1983). It's possible to tell the kernel (on the kernel command line) to not use APIC in which case it will use the old XT-PIC if its available. But since APIC can

have more interrupts than the 15 provided by XT-PIC, there could be problems ??

To see if you have PIC or APIC look at */proc/interrupts*. If you see *XT-PIC* for just irq 2 but *IO-APIC* for the others, it may mean that you have the old XT-PIC but it isn't being currently used. Well, irq 2 is available for communication between two old XT-PICs just in case you might need to use them if you were to disable APIC. There are two XT-PICs since each only supports 8 interrupts.

7.4 Message Signalled Interrupts (MSI)

Another development is Message Signalled Interrupts (MSI) where the interrupt is just a message sent to a special address over the main computer bus (no interrupt lines needed). But the device that sends such a message must first gain control of the main bus so that it can send the interrupt message. Such a message contains more info than just "I'm sending an interrupt". It contains an index for the address of program that needs to be run to service the IRQ. That index, such as 3, would mean for the cpu to find the address it must jump to in the 3rd element of a special table that the cpu knows about.

Since cards must support MSI and many cards don't, it seems that the conventional methods of interrupt hardware support (called INTx) will be around for a long time.

7.5 Sharing PCI Interrupts

PCI interrupts may be shared, meaning that two or more PCI devices will generate the same IRQ. If feasible, it's usually better not to share. Sharing doesn't work right for: 1. very old PCI hardware (before 1995 ??) 2. defective PCI hardware which may have a factory defect (it was made that way). For example, if a PCI device on IRQ9 falsely claims that any IRQ9 was intended for it, then other devices using IRQ9 may wind up having all IRQs they issue ignored since the bad device is falsely claiming their IRQs. With no sharing, this problem is avoided.

For an example of sharing the same IRQ between two PCI devices. see PCI interrupt sharing This sharing ability is built into the hardware and all device drivers are supposed to support it. Note that you usually can't share the same interrupt between the PCI and ISA bus.

7.6 Looking at Routing Tables

Some info is provided by the boot-time messages which may be viewed by typing *dmesg*. The following ways of looking at tables involve software which you may not have (or doesn't exist yet). To check routing where PCI routes to the 16 ISA interrupts use *pirtool* which shows the $PIR routing table. If you have APIC with hard-wired routing (no PIR), use *mptable* to look at the MP table. For routeable APIC, a table is accessed by ACPI _PRT methods (but is there a command-line command to do this?)

7.7 For More Information

Detailed technical information about interrupts is at PCI Interrupts for x86 Machines under FreeBSD. Microsoft has The Importance of Implementing APIC-Based Interrupt Subsystems on Uniprocessor PCs

7.8 PCI Interrupt Linking

Here are some of the details of the PCI interrupt system. Each PCI card (and device mounted on the motherboard) has 4 possible interrupts: INTA#, INTB#, INTC#, INTD#. From now on we will call them just A, B, C, and D. Each has its own pin on the edge connector of a PCI card. Thus for a 7-slot system (for 7 cards) there could be 7 x 4 = 28 different interrupt lines for these cards. Devices built into the motherboard also have additional interrupts. But the specs permit a fewer number of interrupt lines, so some PCI buses seem to be made with only 4 or 8 interrupt lines. This is not too restrictive since interrupts may be shared. For 4 interrupt line (wires, traces, or links) LNKA, LNKB, LNKC, LNKD there is a programmable "interrupt router" chip that routes LNKA, LNKB, LNKC, LNKD to selected IRQs. This routing can be changed by the BIOS or Linux. For example, LNKA may be routed to IRQ5. Suppose we designate the B interrupt from slot 3 as interrupt 3B. Then interrupts 3B and 2A could both be permanently connected to LNKA which is routed to IRQ5. These 2 interrupts: 3B and 2A are permanently shared by hardwiring on the motherboard.

One may type "dmesg" on the command line to see how interrupt lines like LNKA are routed (or linked) to IRQs (*5 means that it's linked to IRQ 5). Look for "PCI Interrupt Link". Note that "link" is used here with two meanings: 1. the linking (routing) of PCI interrupt lines to IRQs. 2. the label of an interrupt line such as LNKB (link B). The interrupt line labels seem to be provided by the Bios ?? and they may have many different names like: LNKC, LNK2, APCF, LUBA, LIDE, etc. Question: When a large number of interrupt lines are shown disabled, do they all physically exist on the motherboard? Or do they just exist only in the ACPI BIOS software so that the BIOS can work with motherboards which have more interrupt lines?

One simple method of connecting (hard-wiring) these lines from PCI devices (such as 3B) to the interrupts LNKA, etc. would be to connect all A interrupts (INTA#) to line LNKA, all B's to LNKB, etc. This method was once used many years ago but it is not a good solution. Here's why. If a card only needs one interrupt, it's required that it use A. If it needs two interrupts, it must use both A and B, etc. Thus INTA# is used much more often than INTD#. So one winds up with an excessive number of interrupts sharing the first line (LNKA connected to all the INTA#). To overcome this problem one may connect them in a more random way so that each of the 4 interrupt lines (LNKA, LNKB, LNKC, LNKD) will share about the same number of actual PCI interrupts.

One method of doing this would be to have wire LNKA share interrupts 1A, 2B, 3C, 4D, 5A, 6B, 7C. This is done by physically connecting wire W to wires 1A, 2B, etc. Likewise wire LNKB could be connected to wires 1B, 2C, 3D, 4A, 5B, 6C, 7D, etc. Then on startup, the BIOS maps the LNKB, LNKA, LNKC, LNKD to IRQs. After that, it writes the IRQ that each device uses into a hardware configuration register in each device. From then on, any program interrogating this register can find out what IRQ the device uses. Note that just writing the IRQ in a register on a PCI card doesn't in any way set the IRQ for that device.

A practical use for this info is that, as a last resort, one may change the IRQs of a PCI card by inserting it in a different slot. In the above example, INTA# of a PCI card will be connected to wire LNKA if the card is inserted into slot 1 (1A maps to LNKA) but INTA# will be connected to wire LNKB when it's inserted into slot 4 (4A maps to LNKB).

A card in a slot may have up to 8 devices on it but there are only 4 PCI interrupts for it (A, B, C, D). This is OK since interrupts may be shared so that each of the 8 devices (if they exist) can have an shared interrupt. The PCI interrupt letter of a device is often fixed and hardwired into the device. The assignment of interrupts is done by either the BIOS or Linux mapping the PCI interrupts to the ISA-like interrupts as mentioned above.

If there are only 4 lines (LNKA, LNKB, LNKC, and LNKD) as in the above example, the mapping choices that the PCI BIOS has are limited. Some motherboards may use more lines and thus have more choices. For example LNKA-LNKH (8 lines). The boot-time messages (and dmesg) may display them and how they are mapped. The BIOS knows about how they are wired.

On the PCI bus, the BIOS (or Linux) assigns IRQs (interrupts) so as to avoid conflicts with the IRQs it knows about on the ISA bus. Sometimes the CMOS BIOS menu may allow one to assign IRQs to PCI cards or to tell the BIOS what IRQs are to be reserved for ISA devices. The assignments are known as a "routing table". In MS Windows it's called "IRQ steering" but this also covers the case of dynamic IRQ routing after boot-time. The BIOS may support it's own IRQ steering.

If your PC uses PCI interrupts which are mapped to ISA interrupts, you right think that interrupts might be slow since the ISA bus was slow. Not really. The ISA Interrupt Controller Chip(s) has a direct interrupt wire going to the CPU so it can get immediate attention. While signals on the old ISA address and data buses are slow to get to the CPU, the IRQ interrupt signals get there very fast.

8. PnP for External and Plug-in Devices

8.1 USB Bus

The USB (Universal Serial Bus) is a high speed bus on an external cable that plugs into a PC. The external bus cable has its own communication protocols and doesn't use any IRQs, I/0 addresses (or other bus-resources). Communication is by packets, something like the Internet, only there are time-slice allocations which prevent any one device from hogging the bus if other devices need it. There are free time slots which allow every device to send a short message to the bus controller without any need for IRQs on the bus.

However, the USB bus controller inside the PC does have an IRQ and an address on the PCI bus (or ISA) which are used for communication between the CPU and all devices on the USB. Thus there's no resource allocations needed for the individual devices on the USB. One could also think of this as all devices on the USB sharing the one interrupt and address. If a device is on the USB it needs a driver that understands the USB.

But each device on the USB does have an IDs, just like cards do on the PCI bus. So Linux likely maintains a table of these IDs so that device drivers can check them to find their device. The USB also support "hot plug". To find out what is on the USB bus, you could use a general hardware detection tool like "discover" or "hwinfo".

8.2 Hot Plug

"Hot plug" is where you plug something into a PC (usually via a cable connection) and it is instantly detected. If required, it is configured with bus-resources. The driver for it is also started, perhaps by loading a module for it. For this to work the hardware used must be designed for hot plugging. One can hot plug certain PCI cards (Cardbus), USB devices, and IEEE 1394 devices (Firewire).

When a new device is detected, it's registers are read so as to get the ID number of the device. Then to find a driver, Linux must maintain a table mapping ID numbers to drivers. It wasn't until kernel 2.4 that such a table existed since Linux once shunned centralized PnP. It's named: MODULE_DEVICE_TABLE.

8.3 Hot Swap

"Hot Swap" is where you remove an old device and then plug in a new device to replace the old one. You have thus "swapped" devices. Now in addition to being able to detect that a new device has been plugged in, the removal of the old device needs to be detected too.

8.4 PnP Finds Devices Plugged Into Serial Ports

External devices that connect to the serial port via a cable (such as external modems) can also be called Plug-and-Play. Since only the serial port itself needs bus-resources (an IRQ and I/O address) there are no bus-resources to allocate to such plug-in devices. In this case, PnP is used only to identify the modem (read it's model code number). This could be important if the modem is a software modem (linmodem) and requires a special driver. There is a special PnP specification for such external serial devices (something connected to the serial port).

Linux doesn't support this yet ?? For a hardware modem, the ordinary serial driver will work OK so there's little need for using the special serial PnP to find a driver. You still need to tell the communications program what port (such as /dev/ttyS1) the modem is on. With PnP you wouldn't need to even do this. With the advent of software modems that have Linux drivers (linmodems), it would be nice to have the appropriate driver install itself automatically via PnP.

9. Error Messages

9.1 Unexpected Interrupt

This means that an interrupt happened that no driver expected. It's unlikely that the hardware issued an interrupt by mistake. It's more likely that the software has a minor bug and doesn't realize that some software did something to cause the interrupt. In many cases you can safely ignore this error message, especially if it only happens once or twice at boot-time. For boot-time messages, look at the messages which are nearby for a clue as to what is going on. For example, if probing is going on, perhaps a probe for a physical device caused that device to issue an interrupt that the driver didn't expect. Perhaps the driver wasn't listening for the correct IRQ number.

9.2 Plug and Play Configuration Error (Dell BIOS)

The BIOS was unable to configure bus-resource. There may be an interrupt conflict which can't be avoided. Dell suggests that you remove some of your non-essential cards and see if it goes away. In one case this problem was due to a defective motherboard.

9.3 isapnp: Write Data Register 0xa79 already used (from logs)

If you use isa-pnp, the IO address 0xa79 must not ever be used by any device. So if other hardware is using 0xa79 when you try to load the isa-pnp module, you'll get this message in your logs and the isa-pnp will exit. One way to try to fix this is to load the isa-pnp module early before other hardware is initialized. For PCMCIA this means to load isa-pnp before running cb modules and service.

9.4 Can't allocate region (PCI)

Here "region" means address range. A PCI device that needs two addresses will have region 0 for the first address and region 1 for the second address needed. Use the command: lspci -vv to see the various resource regions (often just called regions) and whether the address is of type IO or memory. In PCI jargon region 2 is "base address 2" (or "base address register 2"), etc.

10. Interrupt Sharing and Interrupt Conflicts

10.1 Introduction

When two or more devices use the same interrupt line (and the same IRQ number) it's either "Interrupt Sharing" or an "Interrupt Conflict". The PCI bus allows all PCI devices to share interrupts with each other so this is called "sharing". But if an ISA device (or a LPC device ??) uses the same interrupt (IRQ) as some other device (either PCI, ISA, or LPC ??) there is usually an interrupt conflict.

There are exceptions to what's stated above. Some very old PCI devices (pre 1995 ??) do not allow interrupt sharing. Conversely, a few ISA devices have been designed to share interrupts (between two ISA devices ??) but both ISA devices must be designed this way and be driven by software that knows about sharing interrupts. The motherboard must support it too. The following discussion pertains to PCs that have an ISA bus.

A conflict means that when an interrupt happens, no device driver (or the wrong one) may be called and bad things happen like buffer overruns (loss of data). A device may nearly ground its interrupt line when it's not sending its interrupt, thus preventing any other device from using that interrupt wire. That's OK if only that device uses that interrupt. But if a second device tries to use the same interrupt line it can't do so. If this second device also nearly grounds the line when not sending an interrupt, then neither device can use the interrupt. But both Linux and the two devices are unaware of this conflict and merrily send out interrupts anyway that mostly go nowhere and are thus lost.

Interrupt conflicts were common when the IRQs were set by jumpers on cards (ISA bus) because the kernel often didn't know how these jumpers were set. ISA plug-and-play (no jumpers on the cards) helped since the software could change IRQs. The demise of ISA in favor of PCI has nearly eliminated IRQ conflicts. Still, your PC likely has devices on the motherboard (not on a plug-in card) on an ISA bus, a LPC bus, or an X-bus. But the BIOS and the kernel should know how these are set and thus avoid using them for PCI devices, thereby avoiding interrupt conflicts. But there is still a possible interrupt problem with PCI since it could run out of available interrupts, especially on older PCs that only have 16 interrupts.

But IRQ sharing on the PCI bus, while eliminating the conflict problem, has introduced another problem which is less serious: the IRQ balancing problem. If too many high-irq-issuing devices share the same IRQ, it may cause delays in the IRQs getting serviced and can't even result in buffer overruns and other errors. This is not due to congestion on the interrupt line, but it's due to the way that the software determines which device issued the interrupt. PCI interrupt sharing

There are two types of interrupt conflicts. One is a real conflict as described above. In this case interrupts don't work and the device driver keeps trying to control its device and is not aware that interrupts are not working. The second type of interrupt conflict is where a device driver is started but discovers that the interrupt it needs is already in use so it issues an error message and exits. The message could say something like "resource busy", and not clearly state that it was an interrupt problem.

10.2 Real Interrupt Conflict

Both the BIOS and the kernel will not knowingly allow any interrupt conflict, so how can they happen? One way is if someone has put an incorrect IRQ into a configuration file, such as giving a parameter to a module like: irq=9. In this example, suppose the irq of the device is really irq5. Then when another device driver starts up where its device is set to irq5, you have two real devices using irq5 and a real conflict. The kernel

approved of letting the second device use irq5 since it erroneously thought that the first device was using irq9 and that irq5 was free.

There are other cases like this where the kernel fails to know that an irq is in use. One is when an old ISA card with an irq set by a jumper is present, but it's driver hasn't started yet (or it may not even have a driver). Another case is where the BIOS set an irq in the hardware but no linux driver for that hardware ever started and Linux doesn't know about that irq. This can happen even for a PCI card and the irq will show up in *lspci -v* but will not be in the */proc/interrupts* directory and thus not known by the kernel. Is this a bug in the kernel?

What are the symptoms of an interrupt conflict. One might think that the devices will not work at all, but since the addresses are known, the driver does communicate. Interrupts are often used to control the flow of data to and from the device and without interrupts, flow is not controlled. This may mean buffer overruns or even no flow at all since interrupt are used to initiate flow. For a serial modem, the result is extremely slow flow with long pauses and frequent errors. For a sound card it may mean that a word or two is heard and then nothing more.

10.3 No Interrupt Available

This is when a device driver starts but immediately exits in order to avoid an interrupt conflict. It should display or log an error message something like "resource busy".

One case when an ISA device is activated but can't be assigned an interrupt (IRQ) since none are available. Or an interrupt may be available, but it can't be used since the hardware of the device that needs the interrupt doesn't support the interrupt number available (or the motherboard doesn't support it due to "routing" problems, see PCI Interrupts). If the ISA devices use up all the interrupts, then one or more PCI cards may be in conflict since they can't get any IRQs.

Normally, the BIOS will assign interrupts and will not create conflicts. But it may be forced to create conflicts if it runs out of IRQs. This can happen if someone has set up the BIOS to reserve certain IRQs for legacy ISA devices that are not PnP. These settings may be wrong and should be checked out, especially if you're having problems. For example, someone may have reserved an IRQ for an ISA card that has long since been removed from the PC. If you unreserved this IRQ then this IRQ is available and the conflict disappears.

Sometimes the BIOS will solve the problem of an IRQ shortage by using what it calls IRQ 0. There is no such IRQ available since the real IRQ 0 is permanently assigned to the computer's timer. But IRQ 0 here means that the driver should use polling instead of IRQs. This means that the driver frequently checks the device (polls it) to see if the device needs servicing by the interrupt service routine. Of course, this wastes computer time and there's more likelihood of a buffer overrun inside a device since it might not get serviced by the driver promptly enough.

11. Appendix

11.1 Universal Plug and Play (UPnP)

This is actually a sort of network plug-and-play developed by Microsoft but usable by Linux. You plug something into a network and that something doesn't need to be configured provided it will only communicate with other UPnP enabled devices on the network. Here "configure" is used in the broad sense and doesn't mean just configuring bus-resources. One objective is to allow people who know little about networks or

configuring to install routers, gateways, network printers, etc. A major use for UPnP would be in wireless networking.

UPnP uses:

- Simple Service Discovery Protocol to find devices
- General Event Notification Architecture
- Simple Object Access Protocol for controlling devices

This HOWTO doesn't cover UPnP. UPnP for Linux is supported by Intel which has developed software for it. There are other programs which do about the same thing as UPnP. A comparison of some of them is at http://www.cs.umbc.edu/~dchakr1/papers/mcommerce.html A UPnP project for Linux is at SourceForge: UPnP SDK for Linux

11.2 Address Details

There are three types of addresses: main memory addresses, I/O addresses (ports) and configuration addresses. On the PCI bus, configuration addresses constitute a separate address space just like I/O addresses do. Except for the complicated case of ISA configuration addresses, whether or not an address on the bus is a memory address, I/O address, or configuration address depends only on the voltage on other wires (traces) of the bus. For the ISA configuration addresses see ISA Bus Configuration Addresses (Read-Port etc.) for details

Address ranges

The term "address" is sometimes used in this document to mean a contiguous range of addresses. Addresses are in units of bytes, So for example, a serial port at I/O address range 3F8-3FF will often just be referred to by its base address, 3F8. The 3F8 is the location of the first byte in the range (address range). To see the address ranges for various devices, look at /proc/iomem and /proc/ioports.

Address space

To access both I/O and (main) memory address "spaces" the same address bus is used (the wires used for the address are shared). How does the device know whether or not an address which appears on the address bus is a memory address or I/O address? Well, for ISA (for PCI read this too), there are 4 dedicated wires on the bus that convey this sort of information. If a certain one of these 4 wires is asserted, it says that the CPU wants to read from an I/O address, and the main memory ignores the address on the bus. In all, read and write wires exist for both main memory and I/O addresses (4 wires in all).

For the PCI bus it's the same basic idea (also using 4 wires) but it's done a little differently. Instead of only one of the four wires being asserted, a binary number is put on the wires (16 different possibilities). Thus, more info may be conveyed by these 4 wires.. Four of these 16 numbers serve the I/O and memory spaces as in the above paragraph. In addition there is also configuration address space which uses up two more numbers. This leaves 10 more numbers left over for other purposes.

PCI Configuration Address Space

This is different from the IO and memory address spaces because configuration address space is "geographic". Each slot for a card has the slot number as part of the address. This way, Linux (or the BIOS) can address a certain slot and find out what type of card is in that slot. Each device has 64 standard byte-size registers and

some of these hold numbers which can unambiguously identify the device. Since the number of slots is limited as are the number of PCI devices built into motherboard, Linux (or the BIOS) only needs to check a limited number of addresses to find all the PCI devices. If it reads all ones (0xFF in hexadecimal) from the first register of a device, then that means that no device is present. Since there is no card or device to supply all these ones (0xFF) number, the PCI "host bridge" on the motherboard supplies (spoofs) this number for all non-existent device.

The PCI slot number is called (in PCI lingo) the "Device Number" and since a card may have up to 8 devices on it, a "Function Number" from 0-7 identifies which device it is on a PCI card. These numbers are part of the geographic address. Linux programmers call it "pci-slot-name". Thus what Linux calls a "device" is actually a "function" in PCI lingo. The PCI bus number (often 00) also becomes part of the geographic address. For example, 0000:00:0d.2 is PCI bus 0, slot 0d, function 2. For the full geographic address, one must include the double-word number of the device's configuration registers which one wants to access. The leading 0000 (in 1999) were reserved for future use.

How does the CPU designate that a read or write is to a PCI configuration space? It doesn't, at least not directly. Instead when access to configuration space is desired it does a 32-bit (double-word) write to 0cf8-0cfb in IO space and writes the full geographic address there. The PCI host bridge is listening at this address and insures that any next write of data to 0cfc-0cff is put into the specified configuration registers of the specified device. The bridge does this both by sending a special signal to the specified PCI card (or the like) on a dedicated wire that goes only to the slot where the card is plugged in. It also puts bits on the control bus saying that what's on the address bus now is a geographic configuration space address.

Why not make it simple and just have the CPU put bits on the control bus to say that the address on the main bus is a geographic one for PCI configuration? Well, most CPU's are not capable of doing this so the PCI host bridge gets to do it instead.

Range Check (ISA Testing for IO Address Conflicts)

On the ISA bus, there's a method built into each PnP card for checking that there are no other cards that use the same I/O address. If two or more cards use the same IO address, neither card is likely to work right (if at all). Good PnP software should assign bus-resources so as to avoid this conflict, but even in this case a legacy card might be lurking somewhere with the same address.

The test works by a card putting a known test number in its own IO registers. Then the PnP software reads it and verifies that what it reads is the same as the known test number. If not, something is wrong (such as another card with the same address). It repeats the same test with another test number. Since it actually checks the range of IO addresses assigned to the card, it's called a "range check". It could be better called an address-conflict test. If there is an address conflict you get an error message.

Communicating Directly via Memory

Traditionally, most I/O devices used only I/O memory to communicate with the CPU. The device driver, running on the CPU would read and write data to/from the I/O address space and main memory. Unfortunately, this requires two steps. For example, 1. read data from a device (in IO address space) and temporarily store in the CPU; 2. write this data to main memory. A faster way would be for the device itself to put the data directly into main memory. One way to do this is by using ISA DMA Channels or PCI bus mastering. Another way is for the physical device to actually contain some main memory (at high addresses so as not to conflict with main memory chip addresses). This way the device reads and writes directly to it's self-contained main memory without having to bother with DMA or bus mastering. Such a device may also

use IO addresses.

11.3 ISA Bus Configuration Addresses (Read-Port etc.)

These addresses are also known as the "Auto-configuration Ports". For the ISA bus, there is technically no configuration address space, but there is a special way for the CPU to access PnP configuration registers on the PnP cards. For this purpose 3 @ I/O addresses are allocated and each addresses only a single byte (there is no "range"). This is not 3 addresses for each card but 3 addresses shared by all ISA-PnP cards.

These 3 addresses are named read-port, write-port, and address-port. Each port is just one byte in size. Each PnP card has many configuration registers so that just 3 addresses are not even sufficient for the configuration registers on a single card. To solve this problem, each card is assigned a card number (handle) using a technique called "isolation". See ISA Isolation for the complex details.

Then to configure a certain card, its card number (handle) is sent out via the write-port address to tell that card that it is to listen at its address port. All other cards note that this isn't their card number and thus don't listen. Then the address of a configuration register (for that card) is sent to the address-port (for all cards --but only one is listening). Next, data transfer takes place with that configuration register on that card by either doing a read on the read-port or a write on the write-port.

The write-port is always at A79 and the address-port is always at 279 (hex). The read-port is not fixed but is set by the configuration software at some address (in the range 203-3FF) that will hopefully not conflict with any other ISA card. If there is a conflict, it will change the address. All PnP cards get "programmed" with this address. Thus if you use say isapnp to set or check configuration data it must determine this read-port address.

11.4 Interrupts --Details

Serialized Interrupts

It was previously stated that there was a wire for each interrupt. But the serialized interrupt (or serial interrupt) is an exception. A single wire is used for all interrupt which are multiplexed on that wire. Each interrupt has a time slot on the interrupt line. It's used on the LPC bus and is also for the PCI bus, but it's seldom used for PCI ??

DMA

Before going into interrupt details, there is another way for some devices to initiate communication besides sending out an interrupt. This method is a DMA (Direct Memory Access) request to take control of the computer from the CPU for a limited amount of time. On the PCI bus, it uses no "resources". Not all devices are capable of doing DMA. See DMA Channels.

Soft interrupts

There's also another type of interrupt known as a "soft interrupt" which is not covered in this HOWTO and doesn't use any "resources". While a hardware interrupt is generated by hardware, a soft interrupt is initiated by software. There are a couple of ways to do this. One way is for software to tell the CPU to issue an interrupt (an interrupt instruction). Another way is for the software to send messages to other processes so as to interrupt them although it's not clear that this should be called an interrupt. The ksoftirq process, which you may find running on a Linux PC, is a program which does this kind of interrupt for dealing with device

drivers. The device driver starts running due to a hardware interrupt but later on, software interrupts are used for the "bottom half" of the driver's interrupt service routine. Thus, the ksoftirq process is also known as "bottom-half". For more details see the kernel documentation.

Hardware interrupts

Interrupts convey a lot of information but only indirectly. The interrupt request signal (a voltage on a wire) sent by a device just tells a chip called the interrupt controller that a certain device needs attention. The interrupt controller then signals the CPU. The CPU then interrupts whatever it was doing, finds the driver code for this device and runs a part of it known as an "interrupt service routine" (or "interrupt handler"). This "routine" tries to find out what has happened and then deals with the problem. For example, bytes may need to be transferred from/to the device. This program (routine) can easily find out what has happened since the device has registers at addresses known to the driver software (provided the IRQ number and the I/O address of the device has been set correctly). These registers contain status information about the device . The software reads the contents of these registers and by inspecting the contents, finds out what happened and takes appropriate action.

Thus each device driver needs to know what interrupt number (IRQ) to listen for. On the PCI bus (and for some special cases on the ISA bus) it's possible for two (or more) devices to share the same IRQ number. Note that you can't share a PCI interrupt with an ISA interrupt (are there any exceptions ??). When a shared interrupt is issued, the CPU runs all interrupt service routines sequentially for all devices using that interrupt. The first thing such a service routine does is to check its device's registers to see if an interrupt actually happened for its device. If it finds that its device didn't issue an interrupt (a false alarm) then it likely will immediately exit and the next service routine begins for the second device which uses that same interrupt. It checks out the device like described above. This sequence is repeated until the device is found that actually issued the interrupt. All the interrupt routines for an interrupt are said to be constitute a chain. So the chain is traversed until a routine on the chain claims the interrupt by saying in effect: this interrupt was for me. After it handles the interrupt, the interrupt service routines further out on the chain don't run.

The putting of a voltage on the IRQ line is only a request that the CPU be interrupted so it can run a device driver. In almost all cases the CPU is interrupted per the request. But interrupts may be temporarily disabled or prioritized so that in rare cases the actual interrupt of the CPU doesn't happen (or gets delayed). Thus what was above called an "interrupt" is more precisely only an interrupt request and explains why IRQ stands for Interrupt ReQuest.

11.5 How the Device Driver Catches its Interrupt

The previous statement, that device drivers listen for their interrupt, was an oversimplification. Actually it's a chip (or part of a chip) on the motherboard called the "interrupt controller" that listens for all interrupts. When the interrupt controller catches an interrupt, it sends a signal to the CPU to start the appropriate device driver's "interrupt service routine" to handle the interrupt.

There are various types of interrupt controllers. One type is the APIC = Advanced Programmable Interrupt Controller which usually has input pins for many interrupts, including PCI interrupts. Older controllers only have pins for ISA interrupts but they can still handle PCI interrupts since there is a "programmable interrupt router" that converts PCI interrupts to ISA interrupts and routes them to certain pins (= certain IRQs) on the ISA interrupt controller.

11.6 ISA Isolation

This is only for the old ISA bus. Isolation is a complex method of assigning a temporary handle (id number or Card Select Number = CSN) to each PnP device on the ISA bus. Since there are more efficient (but more complex) ways to do this, some might claim that it's a simple method. Only one write address is used for PnP writes to all PnP devices so that writing to this address goes to all PnP device that are listening. This write address is used to send (assign) a unique handle to each PnP device. To assign this handle requires that only one device be listening when the handle is sent (written) to this common address. All PnP devices have a unique serial number which they use for the process of isolation. Doing isolation is something like a game. It's done using the equivalent of just one common bus wire connecting all PnP devices to the isolation program.

For the first round of the "game" all PnP devices listen on this wire and send out simultaneously a sequence of bits to the wire. The allowed bits are either a 1 (positive voltage) or an "open 0" of no voltage (open circuit or tri-state). To do this, each PnP device just starts to sequentially send out its serial number on this wire, voltage (open circuit or tri-state). To do this, each PnP device just starts to sequentially send out its serial number on this wire, bit-by-bit, starting with the high-order bit. If any device sends a 1, a 1 will be heard on the wire by all other devices. If all devices send an "open 0" nothing will be heard on the wire. The object is to eliminate (by the end of this first round) all but highest serial number device. "Eliminate" means to drop out of this round of the game and thus temporarily cease to listen anymore to the wire. (Note that all serial numbers are of the same length.) When there remains only one device still listening, it will be given a handle (card number).

First consider only the high-order bit of the serial number which is put on the wire first by all devices which have no handle yet. If any PnP device sends out a 0 (open 0) but hears a 1, this means that some other PnP device has a higher serial number, so it temporarily drops out of this round. Now the devices remaining in the game (for this round) all have the same leading digit (a 1) so we may strip off this digit and consider only the resulting "stripped serial number" for future participation in this round. Then go to the start of this paragraph and repeat until the entire serial number has been examined for each device (see below for the all-0 case).

Thus it's clear that only cards with the lower serial number get eliminated during a round. But what happens if all devices in the game all send out a 0 as their high-order bit? In this case an "open 0" is sent on the line and all participants stay in the game. If they all have a leading 0 then this is a tie and the 0's are stripped off just like the 1's were in the above paragraph. The game then continues as the next digit (of the serial number) is sent out.

At the end of the round (after the low-order bit of the serial number has been sent out) only one PnP device with the highest serial number remains in the game. It then gets assigned a handle and drops out of the game permanently. Then all the dropouts from the previous round (that don't have a handle yet) reenter the game and a new round begins with one less participant. Eventually, all PnP devices are assigned handles. It's easy to prove that this algorithm works. The actual algorithm is a little more complex than that presented above since each step is repeated twice to ensure reliability and the repeats are done somewhat differently (but use the same basic idea).

Once all handles are assigned, they are used to address each PnP device for sending/reading configuration data. Note that these handles are only used for PnP configuration and are not used for normal communication with the PnP device. When the computer starts up a PnP BIOS will often do such an isolation and then a PnP configuration. After that, all the handles are "lost" so that if one wants to change (or inspect) the configuration again, the isolation must be done over again.

11.7 Bus Mastering and DMA resources

If a bus has bus mastering available, it's unlikely that any resources will be needed for DMA on that bus. For example, the PCI bus doesn't need DMA resources since it has "bus mastering". However, "bus mastering" is often called DMA. But since it's not strictly DMA it needs no DMA resources. The ISA and VESA local bus had no bus mastering. The old MCA and EISA buses did have bus mastering.

11.8 Historical and Obsolete

OSS-Lite Sound Driver

You must give the IO, IRQ, and DMA as parameters to a module or compile them into the kernel. But some PCI cards will get automatically detected. RedHat supplies a program "sndconfig" which detects ISA PnP sound cards and automatically sets up the modules for loading with the detected bus-resources.

ALSA (Advanced Linux Sound Architecture) as of 2000

This will detect the card by PnP methods and then select the appropriate driver and load it. It will also set the bus-resources on an ISA-PnP cards or PCI cards. OSS (Open Sound System) was formerly popular.

MS Windows Notes

Windows NT4 didn't support ISAPNP but had a PNPISA program which one could "use at your own risk". For NT4 users were advised to set "not a PnP OS" in the BIOS so that the BIOS would do the resource configuring. Thus both MS Windows and Linux were in olden days dependent on the BIOS doing the configuring (and still are).

END OF Plug-and-Play-HOWTO