Plug−and−Play−HOWTO

Table of Contents

Table of Contents

Plug−and−Play−HOWTO

David S.Lawyer `mailto:dave@lafn.org`

v1.00, November 2000

Help with understanding and dealing with the complex Plug−and−Play (PnP) issue. How to get PnP to work on your PC (if it doesn't already). It doesn't cover what's called "Universal Plug and Play" (UPnP). See Universal Plug and Play (UPnP)

---

1. Introduction

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

3. The Plug−and−Play (PnP) Solution

4. Setting up a PnP BIOS

5. How to Deal with PnP Cards

6. Tell the Driver the Configuration

7. What Is My Current Configuration?

8. Appendix

---

1. Introduction

1.1 Copyright, Trademarks, Disclaimer, & Credits

Copyright

version.
2. License the derivative work in the spirit of this license or use GPL. Include a copyright notice and at least a pointer to the license used.
3. Give due credit to previous authors and major contributors.

If you're considering making a derived work other than a translation, it's requested that you discuss your plans with the current maintainer.

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) should be assumed to be a trademark). Such trademarks belong to their respective owners.

Credits

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

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 month 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 to implement Plug−and−Play. Nor do I fully understand how PnP is configured by the BIOS (it depends on which BIOS) nor how Windows9x updates the ESCD. Thus this HOWTO is still incomplete and may be inaccurate (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 month or two and will be available to browse and/or download at LDP mirror sites. For a list of mirror sites see: http://linuxdoc.org/mirrors.html. Various formats are available. If you only want to quickly check the date of the latest version look at: http://linuxdoc.org/HOWTO/Plug−and−Play−HOWTO.html. The version you are now reading is: v1.00, November 2000 . New is this version are: "skip file" workaround for reconfiguring under Windows. PCI interrupt details: X, W were transposed, rewrote, mentioned MSI, etc. General Introduction added. Revised "How Linux Does PnP", etc.. Patching kernel to make it PnP no longer feasible. Drivers do more now due to kernel provided functions. .

This starts version 1.00 from the previous version 0.12. This version was long overdue and recognizes that the kernel is doing more in helping device drivers set up PnP. Kernel 2.4 is significantly improved in this respect. There's still a lot of improvements needed in both this HOWTO and the way that Linux does PnP.

1.4 General Introduction. Do you need this HOWTO?

Plug−and−play (PnP) is a system which automatically detects PC devices such as disks, sound cards, ethernet cards, modems, etc. It also does some low−level configuring of them. To be detected by PnP, the device must be designed for PnP. Non−PnP devices (or PnP devices which have been correctly PnP−configured), can often be detected by non−PnP methods.

While the Linux kernel has no centralized plug−and−play system, it does provide programs which various device drivers can use to do their own plug−and−play. Many drivers take advantage of this and find your PnP devices OK. The BIOS hardware of your PC likely may also do some plug−and−play work. 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 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 something about how communication takes place inside the computer.

In this document I mention so many things that can go wrong that one who believes in Murphy's Law (If something can go wrong it will) may become quite alarmed. But for PnP for most people: If something can go wrong it usually doesn't. Remember that sometimes problems which seem to be PnP related are actually due to defective hardware or to hardware that doesn't 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 How a Computer Finds Devices (and conversely)

Oversimplified, Plug−and−Play automatically 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 the following "bus−resources" to both drivers and hardware: I/O addresses, IRQs, DMA channels (ISA bus only), and memory regions. These 4 things are sometimes called 1st order resources. 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 about 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 "files" for such devices 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. Even with PnP fully utilized, much configuring of devices is done by other than PnP. For example, for modem configuration an "init string" is sent to the modem over the I/0 address "channel". This "init string" has nothing to do with PnP although the "channel" used to send it to the modem was allocated by PnP. Setting the speed (and many other parameters) of a serial port is done by sending messages to the device driver from programs run by the user, application programs, or by start−up scripts at boot−time. This configuring also has nothing to do with PnP. Thus when talking about PnP, "configuring" means only a certain type of configuring. While other documentation (such a for MS Windows) simply calls bus−resources "resources", I have used the term "bus−resources" so as to distinguish it from the multitude of other kinds of resources.

2.2 How a Computer Finds Devices (and conversely)

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 cards, modem cards, sound cards, the USB bus, serial and parallel ports, etc. 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.

In olden days most all devices had their own plug−in cards (printed circuit boards). Today, in addition to plug−in cards, many "devices" are small chips permanently mounted on the "motherboard". 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 hda1 (first partition on hard drive a), ttyS0 (the first serial port), eth1 (the second ethernet card), etc. To make matters more complicated, the particular device driver selected, say for eth1, will depend on the type of ethernet card you have. Thus eth1 can't just be assigned to any ethernet driver. It must be assigned to a certain driver that will work for the type of ethernet card you have installed. For modules, some of these assignments might be found in /etc/modules.conf (called "alias") while others may reside in an internal kernel table.

To control a device, the CPU (under the control of the device driver) sends commands (and data) to and reads info 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 almost everything else.

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". 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

PC's have 3 address spaces: I/O, main memory, and configuration (only on the PCI bus). All of these 3 types of addresses share the same bus inside the PC. But the voltage on certain dedicated wires on the PC's bus tells which "space" an address is in: I/O, main memory, (see Memory Regions), or configuration. See Address Details for more details. Devices were originally located in I/O address space although 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. There are two main steps to allocate the I/O addresses (or other bus−resources such as interrupts):

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

The two step process above 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 you must obtain (and write down) this house number. In computers the device hardware must first set the address it will use in a special register and then the device driver must obtain this address. Both of these must be done, either automatically by software or by entering the data manually into some file. Problems occur when only one of them gets done.

For manual PnP configuration some people make the mistake of doing only one of these 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 the serial port hardware doesn't have the address you told setserial (or doesn't have any address set in it) then you're in trouble.

An obvious requirement is that before the device driver can use an address it must be first set on 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).

What was said in the last 2 paragraphs regarding addresses applies with equal force to other bus−resources: IRQs −−Overview and DMA Channels. What these are will be explained in the next 3 sections.

2.4 IRQs −−Overview

After reading this you may read Interrupts −−Details for some 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. 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 have just received a lot of bytes destined for main memory and the device needs to tell its driver to fetch these bytes at once and transfer them from the device's nearly full buffer into main memory. 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 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 (part of the bus) 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 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 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.

2.5 DMA Channels

DMA channels are only for the ISA bus. DMA stands for "Direct Memory Access". This is where a device is allowed to take over the main computer bus from the CPU and transfer bytes directly to main memory. Normally the CPU would make such a transfer in a two step process:

1. reading from the I/O memory space of the device and putting these bytes into the CPU itself

2. writing these bytes from the CPU to main memory
1. With DMA it's usually a one step process of sending the bytes directly from the device to memory

The device must have such 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 PCI bus doesn't really have any DMA but instead it has something even better: bus mastering. It works something like DMA and is sometimes called DMA (for example, hard disk drives that call themselves "UltraDMA"). It 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 allocation of DMA channels for the PCI bus.

When a device on the ISA bus 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 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.

2.6 Memory Ranges

Some devices are assigned address space in main memory. It's often "shared memory" or "memory−mapped I/O". This memory is physically located on the device. When discussing bus−resources it's often just called "memory". In addition to using such "memory", such a device might also use conventional I/O address space.

When you plug in such a card, you are in effect also plugging in a memory module for main memory. 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). This memory can serve as a means of direct data "transfer" between the device and main memory. It's not really a transfer since the device puts data into its own memory on its card which also happens to be in main memory. Both the card and the device driver need to know where it is. The memory address is likely to be very high so that it does not conflict with the lower addresses of the memory chips in your computer.

ROM is different. It is likely a program (perhaps a device driver) which will be used with the device. Hopefully, it may work with Linux and not just 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".

2.7 "Resources" for both Device and Driver

Thus device drivers must be "attached" in some way to the hardware they control. This is done by supplying bus−resources (I/O, Memory, IRQ's, DMA's) to both the physical device and the device driver software. For example, a serial port uses only 2 (out of 4 possible) 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). For the case of jumpers, it's the location of the jumpers themselves that store the bus−resource configuration in the device hardware (on the card, etc.). For the case of PnP, 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.

2.8 The Problem

The architecture of the PC provides only a limited number of IRQ's, DMA channels, I/O address, and memory regions. If there were only several devices and they all had standardized bus−resource (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, etc. Each driver would be programmed with the unique addresses, IRQ, etc. hard−coded into the program. Life would be simple.

But it's not. Not only are there so many different devices today that conflicts are frequent, but one sometimes needs to have more than one of the same type of device. For example, one may want to have a few different disk−drives, a few network cards, etc. For these reasons devices need to have some flexibility so that they can be set to whatever address, IRQ, etc. is needed to avoid conflicts. 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. For example, suppose that you enter IRQ 4 in a configuration file when the device is actually set at IRQ 5. This is another type of bus−resource allocation error.

The allocation of bus−resources, if done correctly, establishes 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 info to the device. It's actually a little more than one−way 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.

2.9 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. Thus PnP is not really needed for them. Even so, there is a PnP specification for such external serial devices.

A PnP operating system will find such an external device and read its model number, etc. Then it may be able to find a device driver for it so that you don't have to tell an application program that you have a certain device on say /dev/ttyS1. Since you should be able to manually inform your application program (via a configuration file, etc.) what serial port the device is on (and possibly what model number it is) you should not really need this "serial port" feature of PnP. I don't think Linux supports it ??

---

3. The Plug−and−Play (PnP) Solution

3.1 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, the configuration is only that of configuring PnP bus−resources and letting the device drivers know about it. In a narrower sense it is just setting bus−resources in the hardware devices. For the case of Linux, it is often just a driver detecting how the bus−resources have been set in its device by the BIOS, etc. "PnP' may just mean the PnP specifications which (among other things). There are long specs for PnP on the resource the ISA bus. The standard PCI specifications (which are not called "PnP") do the same for the PCI bus.

PnP matches up devices with their device drivers and specifies their communication channels. On the ISA bus before Plug−and−Play the bus−resources were formerly set in hardware devices by jumpers. Software drivers were assigned bus−resources by configuration files (or the like) or by probing the for the device at addresses where it's expected to reside. The PCI bus was PnP−like from the beginning but at first it wasn't called PnP. While the PCI bus specifications don't use the term PnP it supports in hardware what today is called PnP.

3.2 How It Works (simplified)

Here's an oversimplified view of how PnP should work. The PnP configuration program (in Linux only the BIOS does this) 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 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 card. It then assigns the card IRQ5 and 1 MB of memory addresses space, starting at address 0xe9000000. It's not always this simple as the card may specify that it can only use certain IRQ numbers (ISA only) 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.

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. Windows9x and PnP BIOSs do this but standard Linux doesn't. Windows9x stores this info in its "Registry" on the hard disk and a PnP BIOS stores it in non−volatile memory in your PC (known as ESCD; see The BIOS's ESCD Database).

While MS Windows is a PnP OS, Linux isn't. But PnP still often works in Linux due to the BIOS doing the configuring of bus−resources and the device drivers finding out (using programs supplied by the Linux kernel) what the BIOS has done. Drivers can also change bus−resource assignments using such programs (but they might take a bus−resource needed by another device). Some device drivers store the last configuration they used and use it the next time the computer is powered on.

If the device hardware remembered their previous configuration, then there wouldn't be any hardware to configure at the next boot−time, but they seem to forget their configuration when the power is turned off. Some devices contain a default configuration (but not necessarily the last one used). Thus a PnP configuration program needs to be run each time the PC is powered on. Also, if a new device has been added, then it too needs to be configured. 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 do this.

3.3 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 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 these devices on its own.

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 informing their drivers). This is what usually happens when running Linux.

3.4 Buses

ISA is the old bus of the old IBM PCs while PCI is a newer and faster bus from Intel. The PCI bus was designed for what is today called PnP. It makes it easy (as compared to the ISA bus) to find out how PnP bus−resources have been assigned to hardware devices. To see what has happened use the commands `lspci` or scanpci (Xwindows) and/or look at `/proc/pci` or `/proc/bus/pci`. The boot−up messages on your display are useful (use shift−PageUp to back up thru them). See Boot−time Messages

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 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.

Eventually, the ISA bus should become extinct. When it does, PnP will be easier since it will be easy to find out how the BIOS has configured the hardware. 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. These needs would be better satisfied if Linux was a PnP operating system.

3.5 How Linux Does PnP

Linux has had a serious problem dealing with PnP and still has a problem but it's not as severe as it once was. Linux still is not really a PnP operating system and seems to mainly rely on and device drivers and the PnP BIOS to configure bus−resources for devices. But the kernel provides help for the drivers in the form of PnP programs they may call on. In many cases the device driver does all the needed configuring. In other cases the BIOS may configure and then the device driver may find out how the BIOS has configured it. The kernel provides the drivers with some 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. Kernel 2.2 could do this only for the PCI bus but Kernel 2.4 has this feature for both the ISA and PCI buses (provided that the PNP options have been selected when compiling the kernel). This by no means guarantees that all drivers will fully and correctly use these features.

In addition, the kernel helps avoid resource conflicts by not allowing two devices 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.

Prior to Kernel 2.4, the standalone program: isapnp was often run to configure and/or get info from PnP devices on the ISA bus. isapnp is still needed for cases where the device driver is not fully PnP for the ISA bus.. There was at least one attempt to make Linux a true PnP operating system. See http://www.astarte.free−online.co.uk. But it never was put into the kernel.

To see what help the kernel may provide to device drivers see the kernel documentation: For the PCI bus 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 are so terse that they will not teach you how to write a driver. For the ISA bus see isapnp.txt and possibly (for kernel 2.4) /usr/include/linux/isapnp.h.

When the PC starts up you may note from the messages on the screen that some Linux device drivers often find their hardware devices (and the bus−resources the BIOS has assigned them). But there are situations that a real PnP operating system could handle better:

- A shortage of bus−resources
- More than one driver for a physical device
- Finding a driver for a detected device (instead of making drivers do the searching)
- Saving a lot of work for the programmers of device drivers

---

4. Setting up a PnP BIOS

When the computer is first turned on, the BIOS runs before the operating system is loaded. Modern BIOSs are PnP and will configure some or all of the PnP devices. 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?

4.1 Do you have a PnP operating system?

In any case the PnP BIOS will PnP−configure the hard−drive, video card, and keyboard to make the system bootable. If you said you had a PnP OS it will leave it up to the operating system (or device drivers) to finish the configuration job. If you said no PnP OS then the BIOS should configure everything. If you only run Linux on your PC, you should probably tell it truthfully that you don't have a PnP operating system If you also run MS Windows on your PC and said it was a PnP OS when you installed Windows, then you might try saying that you have a PnP OS to keep Windows happy. If this causes problems in Linux then you can always change it.

If you say you have a PnP OS then you rely on the Linux device drivers and possibly the program isapnp to take care of the bus−resource configuring. This often works OK but sometimes doesn't. In a few cases it's the other way around. Fibbing to Linux this way actually fixed problems. This could be because the BIOS didn't do it's job right but Linux alone did.

If you tell the BIOS you don't have a PnP OS, then the BIOS will do the configuring itself. Unless you have added new PnP devices, it should use the configuration which it has stored in its non−volatile memory (ESCD). See The BIOS's ESCD Database. If the last session on your computer was with Linux, then there should be no change in configuration. See BIOS Configures PnP. But if the last session was with Windows9x (which is PnP) then Windows could have modified 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. So will the isapnp or PCI Utilities programs.

Interoperability with Windows9x or 2k

If you are running both Linux and Windows on the same PC, how do you answer the BIOS's question: Do you have a PnP OS? Normally (and truthfully) you would say no for Linux and yes for Windows. If you're truthful, everything should work OK. But it's a lot of bother to have to set up the BIOS's CMOS menu manually each time you want to switch OSs. To fix this you need to fib about either Linux or Windows. If you're untruthful everything may not work OK but probably can (with some effort) be made to work OK. If you say that you have a PnP OS, then Linux may work OK if all the drivers and isapnp (if you use it) are able to configure OK. Perhaps updating of the Linux OS and/or drivers will help.

Another solution is to set the CMOS for no PnP OS, including when you start Windows. Now you are fibbing to Windows and since Windows is seemingly much more sophisticated in handling PnP. One would expect Windows9x to be able to cope with with hardware that has been fully configured by the BIOS but it can't. (What about Windows2k ??). One might expect that even if Windows didn't realize that the hardware was already configured, it would set the configuration of the physical devices per it's registry and then then everything would work OK.

But it doesn't seem to happen this way. It seems that Windows may just tell its device drivers what has been stored in the Windows' Registry. But the actual hardware configuration (done by the BIOS) is what was stored in the ESCD and may not be the same as the Registry => trouble. So for Windows to work OK you need to get the Registry to contain the bus−resource configuration which the BIOS creates from the ESCD.

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. If this works for you (and this is the latest version of this HOWTO), let me know as I only have one report of this working out OK.

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 CMOS as you start to boot). Windows will then reinstall the devices, hopefully using the bus−resource settings 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" and continue.

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.

4.2 How are bus−resources to be controlled?

This may involve just deciding how to allocate IRQ and DMA bus−resources. If set to "auto", the BIOS will do the allocation. If set to manual, you manually reserve some IRQ's for use on "legacy" (non−pnp) ISA cards. The BIOS may or may not otherwise know about such legacy cards. The BIOS will only know about these legacy cards if you ran ICU (or the like) under Windows to tell the BIOS about them. If the BIOS knows about them, then try using "auto". If it doesn't know about them, then manually reserve the IRQ's needed for the legacy ISA cards and let the rest be for the BIOS PnP to allocate.

4.3 Reset the configuration?

This 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 devices, then you'll need to run ICA again under DOS/Windows to reestablish this data.

---

5. How to Deal with PnP Cards

5.1 Introduction to Dealing with PnP Cards

Today most all new internal boards (cards) are Plug−and−Play (PnP). There are 5 different methods listed below to cope with PnP (but some may not be feasible in your situation). If the device driver configures it, then you don't need to do anything. If the BIOS configures it, you hope that the driver can find out what the BIOS did (you may need to tell it this in a configuration file or the like).

- Device Driver Configures
- BIOS Configures PnP (For the PCI bus you only need a PCI BIOS, otherwise you need a PnP BIOS)
- Disable PnP by jumpers or DOS/Windows software (but many cards can't do this)
- Isapnp is a program you can always use to configure PnP devices on the ISA bus only
- PCI Utilities is for configuring the PCI bus but the device driver should handle it
- 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 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

5.2 Device Driver Configures

Many device drivers (with the help of code provided by the kernel) will use PnP methods to set the bus−resources in the hardware but only for the device that they control. Since the driver has done the configuring, 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.

One problem with this is that the driver may grab bus−resources that are needed by other devices. It does make it easy for the user but a true PnP Linux kernel might be better. See How Linux Does PnP.

5.3 BIOS Configures PnP

Intro to Using the BIOS to Configure PnP

If you have a PnP BIOS, it can configure the hardware. 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.

Your BIOS must support such configuring and there have been cases where it doesn't do it correctly or completely. An advantage of using the BIOS is that it's simple since in most cases there is nothing to set up (except to tell the BIOS's CMOS menu 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? Another possible advantage is that the BIOS does its work before Linux starts so that all the bus−resources are ready to be used (and found) by the device drivers that start up later.

According to MS it's only optional (not required) that a PnP BIOS be able to PnP−configure the devices (without help from MS Windows). But it seems that most of the ones made after 1996 ?? or so can do it. We should send them thank−you notes if they do it right. They configure both the PCI and ISA buses, but it has been claimed that some older BIOSs can only do the 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 BIOSs may have minimal PnP capabilities and seemingly expects 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. 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 if you use a program such as Linux's isapnp or pci utilities (which doesn't update the ESCD) 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 Windows does this only in special cases. See Using Windows to set ESCD.

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.

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

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. As of early 1999 there wasn't any and now in late 2000 no one has told me about any. Thus one may resort to attempting to use Windows (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 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.

5.4 Disable PnP ?

Many devices are PnP only with no option for disabling PnP. But for some, you may be able to disable 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. There are also some reasons why you might not want to disable 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 quite 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).
5. You may have (or will have) other PnP devices that need configuring so that you'll need to provide for (or learn about) PnP anyway.

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).

5.5 Isapnp (part of isapnptools)

Unfortunately, much of the documentation for isapnp is still difficult to understand unless you know the basics of PnP. This HOWTO should help you understand it as well the FAQ that comes with it. `isapnp` is only for PnP devices on the ISA bus (non−PCI). 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 must edit it manually to choose between various options.

With isapnp there's a danger that a device driver which is built into the kernel may run too early before isapnp has set the address, etc. in the hardware. This results 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.

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. 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. If you use "isapnp" for configuring and have a PnP BIOS, you you may want to tell the BIOS (when you set it up) that you don't have a PnP OS since you may want the BIOS to configure the PCI devices. While the BIOS may also configure the ISA devices, isapnp will redo it.

The terminology used in the /etc/isapnp.conf file may seem odd at first. For example for an I0 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.

5.6 PCI Utilities

The package PCI Utilities (= pciutils, incorrectly called "pcitools"), should let you manually PnP−configure the PCI bus. "lspci" or "scanpci" (Xwindows) lists bus−resources while "setpci" sets resource allocations in the hardware devices. It appears that setpci is mainly intended for use in scripts and presently one needs to know 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.

5.7 Windows Configures

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). But there may be a problem with IRQs for PCI devices. As Windows shuts down 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. Check "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.

5.8 PnP Software/Documents

- Isapnptools homepage
- Patch to make the Linux kernel PnP
- PnP driver project
- PnP Specs. from Microsoft
- Book: PCI System Architecture, 3rd ed. by Tom Shanley +, MindShare 1995. Covers PnP−like features on the PCI bus.
- Book: Plug and Play System Architecture, by Tom Shanley, Mind Share 1995. 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.

---

6. Tell the Driver the Configuration

6.1 Introduction

Just how this is done depends upon the driver. Some drivers have more than one way to find out how their physical device is configured. At one extreme is the case where you must hard−code the bus−resources into the kernel and recompile. At the other extreme, the driver does everything automatically and you have nothing to do. It may even set the bus−resources in the hardware using PnP methods.

In the middle are cases where you run a program to give the resource info to the driver or put the info in a file. In some cases the driver may probe for the device at addresses where it suspects the device resides. It may then try to test various IRQs to see which one works. It may or may not automatically do this. In other cases the driver may use PnP methods to find the device and how the bus−resources have been set, but will not actually set them. It may also look in some of the files in the /proc directory.

One may need to give the bus−resources as a parameter to the kernel or to a loadable module. See /usr/lib/modules_help/descr.gz for a list of possible parameters. The module to load is listed in /etc/modules along with its parameters. In some other case the bus−resources may be given as parameters to the kernel. These are put into the lilo.conf file as append="...". Then the lilo program must be run to save this in the kernel boot code.

While there is great 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 driver documentation (check the kernel documentation tree). Some brief info on a few drivers is presented in the following sections:

6.2 Serial Port Driver: setserial

For the standard serial port driver (not for multiport cards) you use setserial to configure the driver. It is often run from a start−up file. In newer versions there is a /etc/serial.conf file 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 way is used to manually tell the driver the configuration. The other way 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. The driver runs something like `setserial` at start−up but it doesn't probe for IRQs, it just assigns the "standard" IRQ which may be wrong. It does probe for the existence of a port. See Serial−HOWTO for more details.

For PCI serial, the serial driver may detect certain modems and configure the bus−resources.

6.3 Sound Card Drivers

OSS−Lite

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 (likely by using the lspci command or the like). RedHat supplies a program "sndconfig" which detects ISA PnP cards and automatically sets up the modules for loading with the detected bus−resources.

OSS (Open Sound System) and ALSA

These 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 card. You may need to manually intervene to avoid conflicts. For the ALSA driver, support for ISA−PnP is optional and you may use isapnp tools if you want to.

---

7. What Is My Current Configuration?

Here "configuration" means the assignment of PnP bus−resources (addresses, IRQs, and DMAs). There are two parts to this question for each device. Each part should have the same answer.

1. What is the configuration of the device driver software? I.e.: What does the driver think the hardware configuration is?
2. What configuration (if any) is set in the device hardware?

Of course the configuration of the device hardware and its driver should be the same (and it normally is). But if things are not working right, it could be because there's a difference. This means the 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's not easy to determine what has been set in the hardware.

Another problem is that when you view configuration messages on the screen, it's sometimes not clear whether the reported configuration is that of the device driver, the device hardware, or both. If the device driver is assigned a configuration and then checks the hardware out to see if it's configured the same, then the configuration reported by the driver should be that of both the hardware and the driver.

But some drivers which don't do this may accept a configuration that doesn't check out. For example, "setserial" will accept a configuration that doesn't check out and thus may only be telling you the configuration of the driver and not the hardware.

7.1 Boot−time Messages

Some info on configuration may be obtained by reading the messages from the BIOS and 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 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.

The BIOS messages will show the actual hardware configuration at that time, but isapnp, or pci utilities, or device drivers may change it later. The BIOS messages are displayed first before the ones from Linux. As an alternative to eventually using Shift−PageUp to read them, try freezing them by hitting the "Pause" key. Press any key to resume. But once the messages from Linux start to appear, it's too late to use "Pause" since it will not freeze the messages from Linux.

7.2 How Are My Device Drivers Configured?

There may be a programs you can run from the command line (such as "setserial" for serial ports) to determine this. The /proc directory tree is useful. It seems that there are many files buried deep in this tree that contain bus−resource info. Only a couple of them will be mentioned here. /proc/ioports shows the I/O addresses that the drivers use (or try if it's wrong). They might not be set this way in hardware.

/proc/interrupts shows only interrupts currently in use. Many interrupts that have been allocated to drivers don't show at all since they're not currently being 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 unless its in use. Again, just because an interrupt shows up here doesn't mean that it exists in the hardware. A clue that it doesn't exist in hardware will be if it shows that 0 interrupts have been issued by this interrupt. Even if it shows some interrupts have been issued there is no guarantee that they came from the device shown. It could be that some other device which is not currently in use has issued them. A device not in use (per the kernel) may still issue some interrupts for various reasons.

7.3 How Are My Hardware Devices Configured?

It's easy to find out what bus−resources have been assigned to devices on the PCI bus with the "lspci" or "scanpci" commands. For for kernels < 2.2: see `/proc/pci` or `/proc/bus/pci` for later kernels. Note that IRQs for `/proc/pci` are in hexadecimal. Don't bother trying to decipher `/proc/bus/pci/devices` since "lspci" will do that for you.

For the ISA bus 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` "tries" (and finds something there) with the I/O address of the found device. They are not the same. To try to find missing hardware on the ISA bus (both PnP and legacy) try the program "scanport" but be warned that it might hang your PC. It will not show the IRQ nor will it positively identify the hardware.

Messages from the BIOS at boot−time tell you how the hardware configuration was then. If only the BIOS does the configuring, then it should still be the same. 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 is in some configuration file (what the driver thinks). Of course, if the device works fine, then it's likely configured the same as the driver.

Some people use Windows to see how bus−resources have be set up. Unfortunately, since the hardware forgets its bus−resource configuration when powered down, the configuration may not be the same under Linux. It sometimes turns out the same, but Windows and Linux do not necessarily use the same logic in configuring. Many devices have bus−resource preferences that are often adhered to by both Linux and Windows. But don't count on devices being configured the same.

8. Appendix

8.1 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

8.2 Address Details

There are three types of addresses: main memory addresses, I/O addresses 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. Since addresses are in units of bytes, a single address only contains one byte but I/O (and main memory) addresses need more than this. So a range of say 8 bytes is often used for I/O address while the range for main memory addresses allocated to a device is much larger. For a serial port (an I/O device) it's sufficient to give the starting I/O address of the device (such as 3F8) since it's well known that the range of addresses for serial port is only 8 bytes. The starting address is known as the "base address". Sometimes just the word "range" is used to mean "address range".

Address space

For ISA, 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, there are 4 dedicated wires on the bus that convey this information and more. 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. The other 3 wires serve similar purposes. Thus 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 or the four wires being asserted, a binary number is put on the wires (16 different possibilities). Thus more info may be conveyed. 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. Ten extra numbers are left over for other purposes.

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 test number in its own IO registers. Then the PnP software reads it and verifies that it reads the same 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 and need to resolve it yourself.

Communicating Directly via Memory

Traditionally, most I/O devices used only I/O memory to communicate with the CPU. For example, the serial port does this. The device driver, running on the CPU would read and write data to/from the I/O address space and 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 DMA Channels or bus mastering. Another way is to allocate some space in main memory to the device. This way the device reads and writes directly to main memory without having to bother with DMA or bus mastering. Such a device may also use IO addresses.

8.3 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.

8.4 Interrupts −−Details

Interrupts convey a lot of information but only indirectly. The interrupt request signal (a voltage on a wire) 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 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 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 to. 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. When such an interrupt is issued, the CPU runs all interrupt service routines for all devices using that interrupt. The first thing the first service routine does is to check its device 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) it likely will immediately exit and the service routine begins for the second device using that same interrupt, etc, etc.

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 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.

8.5 PCI Interrupts

There are two newer developments in PCI interrupts that are not covered here. They are especially important for cases of more than one CPU per computer. One is the Advanced Programmable Interrupt Controller (APIC). Another 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".

Ordinary PCI interrupts are different than ISA interrupts, but since they are normally mapped to IRQ's they behave in about the same way. One major difference is that the BIOS does this mapping. Under Linux it's not feasible to change it ?? unless the CMOS menu will let you do it. Another major difference is that PCI interrupts may be shared. For example IRQ5 may be shared between two PCI devices. This sharing ability is built into the hardware and all device drivers are supposed to support it. Note that you can't share the same interrupt between the PCI and ISA bus. However, illegal sharing will work provided the devices which are in conflict are not in use at the same time. "In use" here means that a program is running which "opened" the device in its C programming code.

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 may call them just A, B, C, and D. Each has its own pin on the edge connector of a card. Thus for a 7−slot system there could be 7 x 4 = 28 different interrupt lines for the cards. But the specs permit a fewer number of interrupt lines so many PCI buses seem to be made with only 4 interrupt lines. This is not too restrictive since interrupts may be shared. Call these lines (wires or traces) W, X, Y, Z. Suppose we designate the B interrupt from slot 3 as interrupt 3B.

One simple method of connecting these lines to the interrupts would be to connect all A interrupts (INTA#) to line W, all B's to X, etc. This method was used several 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 (W connected to all the INTA#). To overcome this problem one may connect them in a more complicated way so that each of the 4 interrupt lines (W, X, Y, Z) will share about the same number of interrupts.

One method of doing this would be to have wire W share interrupts 1A, 2B, 3C, 4D, 5A, 6B, 7C. This is done by physically connecting wire W to wires 1A, 2B, etc. Likewise wire X could be connected to wires 1B, 2C, 3D, 4A, 5B, 6C, 7D, etc. Then on startup, the BIOS maps the X, W, Y, Z 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.

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 interrupt. The PCI interrupt letter of a device is often fixed and hardwired into the device. The assignment of interrupts is done by the BIOS mapping the ISA interrupts to the PCI interrupts as mentioned above. If there are only 4 lines (W, X, Y, and Z) as in the above example, the choices the PCI BIOS has are limited. Some motherboards may use more lines and thus have more choices. The BIOS knows about how this is wired.

The BIOS assigns IRQs (interrupts) so as to avoid conflicts with the IRQs it knows about on the ISA bus. Sometimes in the CMOS BIOS menu one may allow one to assign IRQs to PCI cards.

You might think that since the PCI is using IRQ's (ISA bus) it might be slow since the ISA bus is 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 ISA address and data buses may be slow to get to the CPU, the IRQ interrupt signals get there almost instantly.

8.6 ISA Isolation

This is only for the 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, 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.

END OF Plug−and−Play−HOWTO