Serial HOWTO

Table of Contents

Serial HOWTO.....1
David S.Lawyer dave@lafn.org original by Greg Hankins.....1
1. Introduction.....1
2. How the Hardware Transfers Bytes.....1
3. Serial Port Basics.....1
4. Is the Serial Port Obsolete?.....1
5. Multiport Serial Boards/Cards/Adapters.....2
6. Configuring the Serial Port.....2
7. Serial Port Devices /dev/ttyS2, etc.....2
8. Interesting Programs You Should Know About.....2
9. Speed (Flow Rate).....2
10. Locking Out Others.....2
11. Communications Programs And Utilities.....3
12. Serial Tips And Miscellany.....3
13. Troubleshooting.....3
14. Interrupt Problem Details.....3
15. What Are UARTs? How Do They Affect Performance?.....3
16. Pinout and Signals.....4
17. Voltage Waveshapes.....4
18. Other Serial Devices (not async EIA−232).....4
19. Other Sources of Information.....4
20. Appendix: Obsolete Hardware (prior to 1990) Info.....4
1. Introduction.....4
1.1 Copyright, Disclaimer, & Credits.....5
Copyright.....5
Disclaimer.....5
Trademarks.....5
Credits.....5
1.2 New Versions of this Serial−HOWTO.....6
1.3 Related HOWTO's re the Serial Port.....6
1.4 Feedback.....6
1.5 What is a Serial Port?.....6
2. How the Hardware Transfers Bytes.....7
2.1 Transmitting.....7
2.2 Receiving.....8
2.3 The Large Serial Buffers.....8
3. Serial Port Basics.....9
3.1 What is a Serial Port ?.....9
Intro to Serial.....9
Pins and Wires.....9
RS−232 or EIA−232, etc......10
3.2 IO Address & IRQ.....10
3.3 Names: ttyS0, ttyS1, etc.....10
3.4 Interrupts.....11
3.5 Data Flow (Speeds).....11
3.6 Flow Control.....12
Example of Flow Control.....12
Symptoms of No Flow Control.....13

Table of Contents

Hardware vs. Software Flow Control....13
3.7 Data Flow Path; Buffers....13
3.8 Complex Flow Control Example....14
3.9 Serial Driver Module....15
4. Is the Serial Port Obsolete?....15
4.1 Introduction....15
4.2 EIA−232 Cable Is Low Speed & Short Distance....15
4.3 Inefficient Interface to the Computer....16
5. Multiport Serial Boards/Cards/Adapters....16
5.1 Intro to Multiport Serial....16
5.2 Making multiport devices in the /dev directory....17
5.3 Standard PC Serial Cards....17
5.4 Dumb Multiport Serial Boards (with standard UART chips)....18
5.5 Intelligent Multiport Serial Boards....19
5.6 Unsupported Multiport Boards....21
6. Configuring the Serial Port....21
6.1 PCI Bus Support Underway....22
6.2 Configuring Overview....22
6.3 Common mistakes made re low−level configuring....23
6.4 I/O Address & IRQ: Boot−time messages....23
6.5 What is the current IO address and IRQ of my Serial Port ?....24
What does the device driver think?....25
What is set in my serial port hardware ?....25
What is set in my PnP serial port hardware ?....26
6.6 Choosing Serial IRQs....26
IRQ 0 is not an IRQ....26
Interrupt sharing and Kernels 2.2+....26
What IRQs to choose?....27
6.7 Choosing Addresses −−Video card conflict with ttyS3....27
6.8 Set IO Address & IRQ in the hardware (mostly for PnP)....28
Using a PnP BIOS to I0−IRQ Configure....28
6.9 Giving the IRQ and IO Address to Setserial....29
6.10 High−level Configuring: stty, etc.....29
Configuring Flow Control: Hardware Flow Control is Best....29
7. Serial Port Devices /dev/ttyS2, etc....30
7.1 Serial Port Device Names & Numbers....30
7.2 Link ttySN to /dev/modem ?....30
7.3 Which Connector on the Back of my PC is ttyS1, etc?....31
Inspect the connectors....31
Send bytes to the port....31
Connect a device to the connector....32
Missing connectors....32
7.4 Creating Devices In the /dev directory....32
8. Interesting Programs You Should Know About....32
8.1 Serial Monitoring/Diagnostics Programs....32
8.2 Changing Interrupt Priority....33
8.3 What is Setserial ?....33
Introduction....33

Table of Contents

Probing.....34
Boot−time Configuration.....35
Configuration Scripts/Files.....35
Edit a script (required prior to version 2.15).....35
New configuration method using /etc/serial.conf.....36
IRQs.....37
8.4 Stty.....37
Introduction.....38
Using stty at a "foreign" terminal.....38
Old redirection method.....39
Two interfaces at a terminal.....39
Where to put the stty command ?.....40
8.5 What is isapnp ?.....40
9. Speed (Flow Rate).....40
9.1 Can't Set a High Enough Speed.....40
How speed is set in hardware: the divisor and baud_base.....41
Work−arounds for setting speed.....41
Crystal frequency is not baud_base.....41
9.2 Higher Serial Throughput.....41
10. Locking Out Others.....42
10.1 Introduction.....42
10.2 Lock−Files.....42
10.3 Change Owners, Groups, and/or Permissions of Device Files.....43
11. Communications Programs And Utilities.....43
11.1 List of Software.....43
11.2 kermit and zmodem.....44
12. Serial Tips And Miscellany.....44
12.1 Serial Module.....44
12.2 Serial Console (console on the serial port).....44
12.3 Line Drivers.....44
12.4 Known Defective Hardware.....44
Avoiding IO Address Conflicts with Certain Video Boards.....44
Problem with AMD Elan SC400 CPU (PC−on−a−chip).....45
13. Troubleshooting.....45
13.1 Serial Electrical Test Equipment.....45
Breakout Gadgets, etc.....45
Measuring Voltages.....45
Taste Voltage.....46
13.2 Serial Monitoring/Diagnostics.....46
13.3 (The following subsections are in both the Serial and Modem HOWTOs).....46
13.4 My Serial Port is Physically There but Can't be Found.....46
13.5 Extremely Slow: Text appears on the screen slowly after long delays.....47
13.6 Somewhat Slow: I expected it to be a few times faster.....47
13.7 The Startup Screen Show Wrong IRQs for the Serial Ports.....47
13.8 "Cannot open /dev/ttyS?: Permission denied".....48
13.9 "Operation not supported by device" for ttyS?.....48
13.10 "Cannot create lockfile. Sorry".....48
13.11 "Device /dev/ttyS? is locked.".....48

Table of Contents

13.12 "/dev/tty? Device or resource busy" 49
13.13 "Input/output error" from setserial or stty.....49
13.14 Overrun errors on serial port.....50
13.15 Troubleshooting Tools.....50
14. Interrupt Problem Details.....50
14.1 Types of interrupt problems.....50
14.2 Symptoms of Mis−set or Conflicting Interrupts.....51
14.3 Mis−set Interrupts.....51
14.4 Interrupt Conflicts.....52
14.5 Resolving Interrupt Problems.....53
15. What Are UARTs? How Do They Affect Performance? 53
15.1 Introduction to UARTS.....53
15.2 Two Types of UARTs.....53
15.3 FIFOs 54
15.4 UART Model Numbers.....55
16. Pinout and Signals 55
16.1 Pinout.....55
16.2 Signals May Have No Fixed Meaning.....55
16.3 Cabling Between Serial Ports 56
16.4 RTS/CTS and DTR/DSR Flow Control.....56
The DTR and DSR Pins.....57
16.5 Preventing a Port From Opening.....57
17. Voltage Waveshapes.....58
17.1 Voltage for a Bit.....58
17.2 Voltage Sequence for a Byte.....58
17.3 Parity Explained.....58
17.4 Forming a Byte (Framing).....59
17.5 How "Asynchronous" is Synchronized.....59
18. Other Serial Devices (not async EIA−232) 60
18.1 Successors to EIA−232.....60
18.2 EIA−422−A (balanced) and EIA−423−A (unbalanced).....60
18.3 EIA−485.....60
18.4 EIA−530.....60
18.5 EIA−612/613 61
18.6 The Universal Serial Bus (USB).....61
18.7 Firewire.....61
18.8 Synchronization & Synchronous 61
Defining Asynchronous vs Synchronous.....61
Synchronous Communication.....62
19. Other Sources of Information 62
19.1 Books 62
19.2 Serial Software.....62
19.3 Related Linux Documents.....63
19.4 Usenet newsgroups: 63
19.5 Serial Mailing List.....63
19.6 Internet.....63
20. Appendix: Obsolete Hardware (prior to 1990) Info.....64
20.1 Replacing obsolete UARTS.....64

Serial HOWTO

David S.Lawyer `dave@lafn.org` original by Greg Hankins

v2.10, February 2001

This document describes serial port features other than those which should be covered by Modem−HOWTO, PPP−HOWTO, Serial−Programming−HOWTO, or Text−Terminal−HOWTO. It does not cover the Universal Serial Bus (see the kernel documentation for USB). It lists info on multiport serial cards. It contains technical info about the serial port itself in more detail than found in the above HOWTOs and should be best for troubleshooting when the problem is the serial port itself. If you are dealing with a Modem, PPP (used for Internet access on a phone line), or a Text−Terminal, those HOWTOs should be consulted first.

---

1. Introduction

- 1.1 Copyright, Disclaimer, & Credits
- 1.2 New Versions of this Serial−HOWTO
- 1.3 Related HOWTO's re the Serial Port
- 1.4 Feedback
- 1.5 What is a Serial Port?

2. How the Hardware Transfers Bytes

- 2.1 Transmitting
- 2.2 Receiving
- 2.3 The Large Serial Buffers

3. Serial Port Basics

- 3.1 What is a Serial Port ?
- 3.2 IO Address & IRQ
- 3.3 Names: ttyS0, ttyS1, etc.
- 3.4 Interrupts
- 3.5 Data Flow (Speeds)
- 3.6 Flow Control
- 3.7 Data Flow Path; Buffers
- 3.8 Complex Flow Control Example
- 3.9 Serial Driver Module

4. Is the Serial Port Obsolete?

- 4.1 Introduction
- 4.2 EIA−232 Cable Is Low Speed & Short Distance
- 4.3 Inefficient Interface to the Computer

5. Multiport Serial Boards/Cards/Adapters

- 5.1 Intro to Multiport Serial
- 5.2 Making multiport devices in the /dev directory
- 5.3 Standard PC Serial Cards
- 5.4 Dumb Multiport Serial Boards (with standard UART chips)
- 5.5 Intelligent Multiport Serial Boards
- 5.6 Unsupported Multiport Boards

6. Configuring the Serial Port

- 6.1 PCI Bus Support Underway
- 6.2 Configuring Overview
- 6.3 Common mistakes made re low−level configuring
- 6.4 I/O Address & IRQ: Boot−time messages
- 6.5 What is the current IO address and IRQ of my Serial Port ?
- 6.6 Choosing Serial IRQs
- 6.7 Choosing Addresses −−Video card conflict with ttyS3
- 6.8 Set IO Address & IRQ in the hardware (mostly for PnP)
- 6.9 Giving the IRQ and IO Address to Setserial
- 6.10 High−level Configuring: stty, etc.

7. Serial Port Devices /dev/ttyS2, etc.

- 7.1 Serial Port Device Names & Numbers
- 7.2 Link ttySN to /dev/modem ?
- 7.3 Which Connector on the Back of my PC is ttyS1, etc?
- 7.4 Creating Devices In the /dev directory

8. Interesting Programs You Should Know About

- 8.1 Serial Monitoring/Diagnostics Programs
- 8.2 Changing Interrupt Priority
- 8.3 What is Setserial ?
- 8.4 Stty
- 8.5 What is isapnp ?

9. Speed (Flow Rate)

- 9.1 Can't Set a High Enough Speed
- 9.2 Higher Serial Throughput

10. Locking Out Others

- 10.1 Introduction
- 10.2 Lock−Files
- 10.3 Change Owners, Groups, and/or Permissions of Device Files

11. Communications Programs And Utilities

- 11.1 List of Software
- 11.2 kermit and zmodem

12. Serial Tips And Miscellany

- 12.1 Serial Module
- 12.2 Serial Console (console on the serial port)
- 12.3 Line Drivers
- 12.4 Known Defective Hardware

13. Troubleshooting

- 13.1 Serial Electrical Test Equipment
- 13.2 Serial Monitoring/Diagnostics
- 13.3 (The following subsections are in both the Serial and Modem HOWTOs)
- 13.4 My Serial Port is Physically There but Can't be Found
- 13.5 Extremely Slow: Text appears on the screen slowly after long delays
- 13.6 Somewhat Slow: I expected it to be a few times faster
- 13.7 The Startup Screen Show Wrong IRQs for the Serial Ports.
- 13.8 "Cannot open /dev/ttyS?: Permission denied"
- 13.9 "Operation not supported by device" for ttyS?
- 13.10 "Cannot create lockfile. Sorry"
- 13.11 "Device /dev/ttyS? is locked."
- 13.12 "/dev/tty? Device or resource busy"
- 13.13 "Input/output error" from setserial or stty
- 13.14 Overrun errors on serial port
- 13.15 Troubleshooting Tools

14. Interrupt Problem Details

- 14.1 Types of interrupt problems
- 14.2 Symptoms of Mis−set or Conflicting Interrupts
- 14.3 Mis−set Interrupts
- 14.4 Interrupt Conflicts
- 14.5 Resolving Interrupt Problems

15. What Are UARTs? How Do They Affect Performance?

- 15.1 Introduction to UARTS
- 15.2 Two Types of UARTs
- 15.3 FIFOs
- 15.4 UART Model Numbers

16. Pinout and Signals

- 16.1 Pinout
- 16.2 Signals May Have No Fixed Meaning
- 16.3 Cabling Between Serial Ports
- 16.4 RTS/CTS and DTR/DSR Flow Control
- 16.5 Preventing a Port From Opening

17. Voltage Waveshapes

- 17.1 Voltage for a Bit
- 17.2 Voltage Sequence for a Byte
- 17.3 Parity Explained
- 17.4 Forming a Byte (Framing)
- 17.5 How "Asynchronous" is Synchronized

18. Other Serial Devices (not async EIA−232)

- 18.1 Successors to EIA−232
- 18.2 EIA−422−A (balanced) and EIA−423−A (unbalanced)
- 18.3 EIA−485
- 18.4 EIA−530
- 18.5 EIA−612/613
- 18.6 The Universal Serial Bus (USB)
- 18.7 Firewire
- 18.8 Synchronization & Synchronous

19. Other Sources of Information

- 19.1 Books
- 19.2 Serial Software
- 19.3 Related Linux Documents
- 19.4 Usenet newsgroups:
- 19.5 Serial Mailing List
- 19.6 Internet

20. Appendix: Obsolete Hardware (prior to 1990) Info

- 20.1 Replacing obsolete UARTS

---

1. Introduction

This HOWTO covers basic info on the Serial Port and multiport serial cards. Information specific to modems and text−terminals has been moved to Modem−HOWTO and Text−Terminal−HOWTO. Info on getty (the program that runs the login process or the like) has been also moved to these HOWTOs since mgetty and uugetty are best for modems while agetty is best for text−terminals. If you are dealing with a modem, text terminal, or printer, then you may not need to consult this HOWTO. But if you are using the serial port for some other device, using a multiport serial card, trouble−shooting the serial port itself, or want to understand more technical details of the serial port, then you may want to use this HOWTO as well as some of the other HOWTOs. (See Related HOWTO's) This HOWTO lists info on various multiport serial cards since they may be used for either modems or text−terminals. This HOWTO addresses Linux running on PCs (ISA or PCI buses), although it might be valid for other architectures.

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

Credits

Most of the original Serial−HOWTO was written by Greg Hankins. mailto:gregh@twoguys.org H e also rewrote many contributions by others in order to maintain continuity in the writing style and flow. He wrote: ``Thanks to everyone who has contributed or commented, the list of people has gotten too long to list (somewhere over one hundred). Special thanks to Ted Ts'o for answering questions about the serial drivers.'' Approximately half of v2.00 was from Greg Hankins HOWTO and the other half is by David Lawyer. Ted Ts'o has continued to be helpful.

1.2 New Versions of this Serial−HOWTO

New versions of the Serial−HOWTO will be available to browse and/or download at LDP mirror sites. For a list of mirror sites see: http://metalab.unc.edu/LDP/mirrors.html. Various formats are available. If you only want to quickly check the date of the latest version look at http://metalab.unc.edu/LDP/HOWTO/Serial−HOWTO.html and compare it to this version: v2.10 February 2001 . New in this version is: v2.10 EIA−485, frame errors on networks, gkermit, firewire

1.3 Related HOWTO's re the Serial Port

Modems, Text−Terminals, some printers, and other peripherals often use the serial port. Get these HOWTOs from the nearest mirror site as explained above.

- `Modem−HOWTO` is about installing and configuring modems
- `Printing−HOWTO` has info on using a serial printer
- `Serial−Programming−HOWTO` helps you write C programs (or parts of them) that read and write to the serial port and/or check/set its state. A new version has been written by Vern Hoxie but not submitted. A copy is at Internet.
- `Text−Terminal−HOWTO` is about how they work, how to install configure, and repair them. It includes a section on "Make a Terminal the Console" which is useful for using a remote terminal to control a server (via the serial port).

1.4 Feedback

Please send me any questions, comments, suggestions, or additional material. I'm always eager to hear about what you think about this HOWTO. I'm also always on the lookout for improvements! Tell me exactly what you don't understand, or what could be clearer. You can reach me via email at `mailto:dave@lafn.org (David Lawyer).`

1.5 What is a Serial Port?

The conventional serial port (not the newer USB port, or HSSI port) is a very old I/O port. Almost all PC's have them. But Macs (Apple Computer) after mid 1998 (with colored cases) only have the USB port. It's possible, however, to put a conventional serial port device on the USB.

The common specification for the conventional serial port is RS−232 (or EIA−232). The connector for the serial port is often seen as one or two 9−pin connectors (in some cases 25−pin) on the back of a PC. But the serial port is is more than just that. It includes the associated electronics which must produce signals conforming to the EIA−232 specification. See Voltage Waveshapes. One pin is used to send out data bytes and another to receive data bytes. Another pin is a common signal ground. The other "useful" pins are used mainly for signalling purposes with a steady negative voltage meaning "off" and a steady positive voltage meaning "on".

The UART (Universal Asynchronous Receiver−Transmitter) chip does most of the work. Today, the functionality of this chip is usually built into another chip. See What Are UARTs? These have improved over time and old models (several years old) are now obsolete.

The serial port was originally designed for connecting modems but it's used to connect many other devices also such as mice, text−terminals, some printers, etc. to a computer. You just plug these devices into the serial port using the correct cable. Many internal modem cards have a built−in serial port so when you install one inside your PC it's as if you just installed another serial port in your PC.

2. How the Hardware Transfers Bytes

Below is an introduction to the topic, but for a more advanced treatment of it see FIFOs.

2.1 Transmitting

Transmitting is sending bytes out of the serial port away from the computer. Once you understand transmitting, receiving is easy to understand since it's similar. The first explanation given here will be grossly oversimplified. Then more detail will be added in later explanations. When the computer wants to send a byte out the serial port (to the external cable) the CPU sends the byte on the bus inside the computer to the I/O address of the serial port. The serial port takes the byte, and sends it out one bit at a time (a serial bit−stream) on the transmit pin of the serial cable connector. For what a bit (and byte) look like electrically see Voltage Waveshapes.

Here's a replay of the above in a little more detail (but still very incomplete). Most of the work at the serial port is done by the UART chip (or the like). To transmit a byte, the serial device driver program (running on the CPU) sends a byte to the serial port"s I/O address. This byte gets into a 1−byte "transmit shift register" in the serial port. From this shift register bits are taken from the byte one−by−one and sent out bit−by−bit on the serial line. Then when the last bit has been sent and the shift register needs another byte to send it could just ask the CPU to send it another byte. Thus would be simple but it would likely introduce delays since the CPU might not be able to get the byte immediately. After all, the CPU is usually doing other things besides just handling the serial port.

A way to eliminate such delays is to arrange things so that the CPU gets the byte before the shift register needs it and stores it in a serial port buffer (in hardware). Then when the shift register has sent out its byte and needs a new byte immediately, the serial port hardware just transfers the next byte from its own buffer to the shift register. No need to call the CPU to fetch a new byte.

The size of this serial port buffer was originally only one byte, but today it is usually 16 bytes (more in higher priced serial ports). Now there is still the problem of keeping this buffer sufficiently supplied with bytes so that when the shift register needs a byte to transmit it will always find one there (unless there are no more bytes to send). This is done by contacting the CPU using an interrupt.

First we'll explain the case of the old fashioned one−byte buffer, since 16−byte buffers work similarly (but are more complex). When the shift register grabs the byte out of the buffer and the buffer needs another byte, it sends an interrupt to the CPU by putting a voltage on a dedicated wire on the computer bus. Unless the CPU is doing something very important, the interrupt forces it to stop what it was doing and start running a program which will supply another byte to the port's buffer. The purpose of this buffer is to keep an extra byte (waiting to be sent) queued in hardware so that there will be no gaps in the transmission of bytes out the serial port cable.

Once the CPU gets the interrupt, it will know who sent the interrupt since there is a dedicated interrupt wire for each serial port (unless interrupts are shared). Then the CPU will start running the serial device driver which checks registers at I/0 addresses to find out what has happened. It finds out that the serial's transmit buffer is empty and waiting for another byte. So if there are more bytes to send, it sends the next byte to the serial port's I/0 address. This next byte should arrive when the previous byte is still in the transmit shift register and is still being transmitted bit−by−bit.

In review, when a byte has been fully transmitted out the transmit wire of the serial port and the shift register is now empty the following 3 things happen almost simultaneously:

1. The next byte is moved from the transmit buffer into the transmit shift register
2. The transmission of this new byte (bit−by−bit) begins
3. Another interrupt is issued to tell the device driver to send yet another byte to the now empty transmit buffer

Thus we say that the serial port is interrupt driven. Each time the serial port issues an interrupt, the CPU sends it another byte. Once a byte has been sent to the transmit buffer by the CPU, then the CPU is free to pursue some other activity until it gets the next interrupt. The serial port transmits bits at a fixed rate which is selected by the user (or an application program). It's sometimes called the baud rate. The serial port also adds extra bits to each byte (start, stop and perhaps parity bits) so there are often 10 bits sent per byte. At a rate (also called speed) of 19,200 bits per second (bps), there are thus 1,920 bytes/sec (and also 1,920 interrupts/sec).

Doing all this is a lot of work for the CPU. This is true for many reasons. First, just sending one 8−bit byte at a time over a 32−bit data bus (or even 64−bit) is not a very efficient use of bus width. Also, there is a lot of overhead in handing each interrupt. When the interrupt is received, the device driver only knows that something caused an interrupt at the serial port but doesn't know that it's because a character has been sent. The device driver has to make various checks to find out what happened. The same interrupt could mean that a character was received, one of the control lines changed state, etc.

A major improvement has been the enlargement of the buffer size of the serial port from 1−byte to 16−bytes. This means that when the CPU gets an interrupt it gives the serial port up to 16 new bytes to transmit. This is fewer interrupts to service but data must still be transferred one byte at a time over a wide bus. The 16−byte buffer is actually a FIFO (First In First Out) queue and is often called a FIFO. See FIFOs for details about the FIFO along with a repeat of some of the above info.

2.2 Receiving

Receiving bytes by a serial port is similar to sending them only it's in the opposite direction. It's also interrupt driven. For the obsolete type of serial port with 1−byte buffers, when a byte is fully received from the external cable it goes into the 1−byte receive buffer. Then the port gives the CPU an interrupt to tell it to pick up that byte so that the serial port will have room for storing the next byte which is currently being received. For newer serial ports with 16−byte buffers, this interrupt (to fetch the bytes) may be sent after 14 bytes are in the receive buffer. The CPU then stops what it was doing, runs the interrupt service routine, and picks up 14 to 16 bytes from the port. For an interrupt sent when the 14th byte has been received, there could be 16 bytes to get if 2 more bytes have arrived since the interrupt. But if 3 more bytes should arrive (instead of 2), then the 16−byte buffer will overrun. It also may pick up less than 14 bytes by setting it that way or due to timeouts. See FIFOs for more details.

2.3 The Large Serial Buffers

We've talked about small 16−byte serial port hardware buffers but there are also much larger buffers in main memory. When the CPU takes some bytes out of the receive buffer of the hardware, it puts them into a much larger (say 8k−byte) receive buffer in main memory. Then a program that is getting bytes from the serial port takes the bytes it's receiving out of that large buffer (using a "read" statement in the program). A similar situation exists for bytes that are to be transmitted. When the CPU needs to fetch some bytes to be transmitted it takes them out of a large (8k−byte) transmit buffer in main memory and puts them into the small 16−byte transmit buffer in the hardware.

3. Serial Port Basics

You don't have to understand the basics to use the serial port But understanding it may help to determine what is wrong if you run into problems. This section not only presents new topics but also repeats some of what was said in the previous section How the Hardware Transfers Bytes but in greater detail.

3.1 What is a Serial Port ?

Intro to Serial

The serial port is an I/O (Input/Output) device.

An I/O device is just a way to get data into and out of a computer. There are many types of I/O devices such as serial ports, parallel ports, disk drive controllers, ethernet boards, universal serial buses, etc. Most PC's have one or two serial ports. Each has a 9−pin connector (sometimes 25−pin) on the back of the computer. Computer programs can send data (bytes) to the transmit pin (output) and receive bytes from the receive pin (input). The other pins are for control purposes and ground.

The serial port is much more than just a connector. It converts the data from parallel to serial and changes the electrical representation of the data. Inside the computer, data bits flow in parallel (using many wires at the same time). Serial flow is a stream of bits over a single wire (such as on the transmit or receive pin of the serial connector). For the serial port to create such a flow, it must convert data from parallel (inside the computer) to serial on the transmit pin (and conversely).

Most of the electronics of the serial port is found in a computer chip (or a part of a chip) known as a UART. For more details on UARTs see the section

What Are UARTS? But you may want to finish this section first so that you will hopefully understand how the UART fits into the overall scheme of things.

Pins and Wires

Old PC's used 25 pin connectors but only about 9 pins were actually used so today most connectors are only 9−pin. Each of the 9 pins usually connects to a wire. Besides the two wires used for transmitting and receiving data, another pin (wire) is signal ground. The voltage on any wire is measured with respect to this ground. Thus the minimum number of wires to use for 2−way transmission of data is 3. Except that it has been known to work with no signal ground wire but with degraded performance and sometimes with errors.

There are still more wires which are for control purposes (signalling) only and not for sending bytes. All of these signals could have been shared on a single wire, but instead, there is a separate dedicated wire for every type of signal. Some (or all) of these control wires are called "modem control lines". Modem control wires are either in the asserted state (on) of +12 volts or in the negated state (off) of −12 volts. One of these wires is to signal the computer to stop sending bytes out the serial port cable. Conversely, another wire signals the device attached to the serial port to stop sending bytes to the computer. If the attached device is a modem, other wires may tell the modem to hang up the telephone line or tell the computer that a connection has been made or that the telephone line is ringing (someone is attempting to call in). See section Pinout and Signals for more details.

RS−232 or EIA−232, etc.

The serial port (not the USB) is usually a RS−232−C, EIA−232−D, or EIA−232−E. These three are almost the same thing. The original RS (Recommended Standard) prefix became EIA (Electronics Industries Association) and later EIA/TIA after EIA merged with TIA (Telecommunications Industries Association). The EIA−232 spec provides also for synchronous (sync) communication but the hardware to support sync is almost always missing on PC's. The RS designation is obsolete but is still widely used. EIA will be used in this howto. Some documents use the full EIA/TIA designation. For info on other (non−EIA−232) serial ports see the section Other Serial Devices (not async EIA−232)

3.2 IO Address & IRQ

Since the computer needs to communicate with each serial port, the operating system must know that each serial port exists and where it is (its I/O address). It also needs to know which wire (IRQ number) the serial port must use to request service from the computer's CPU. It requests service by sending an interrupt on this wire. Thus every serial port device must store in its non−volatile memory both its I/O address and its Interrupt ReQuest number: IRQ. See Interrupts. For the PCI bus it doesn't work exactly this way since the PCI bus has its own system of interrupts. But since the PCI−aware BIOS sets up chips to map these PCI interrupts to IRQs, it seemingly behaves just as described above except that sharing of interrupts is allowed (2 or more devices may use the same IRQ number).

I/O addresses are not the same as memory addresses. When an I/O addresses is put onto the computer's address bus, another wire is energized. This both tells main memory to ignore the address and tells all devices which have I/O addresses (such as the serial port) to listen to the address to see if it matches the device's. If the address matches, then the I/O device reads the data on the data bus.

3.3 Names: ttyS0, ttyS1, etc.

The serial ports are named ttyS0, ttyS1, etc. (and usually correspond respectively to COM1, COM2, etc. in DOS/Windows). The /dev directory has a special file for each port. Type "ls /dev/ttyS*" to see them. Just because there may be (for example) a ttyS3 file, doesn't necessarily mean that there exists a physical serial port there.

Which one of these names (ttyS0, ttyS1, etc.) refers to which physical serial port is determined as follows. The serial driver (software) maintains a table showing which I/O address corresponds to which ttyS. This mapping of names (such as ttyS1) to I/O addresses (and IRQ's) may be both set and viewed by the "setserial" command. See What is Setserial. This does `not` set the I/O address and IRQ in the hardware itself (which is set by jumpers or by plug−and−play software). Thus what physical port corresponds to say ttyS1 depends both on what the serial driver thinks (per setserial) and what is set in the hardware. If a mistake has been made, the physical port may not correspond to any name (such as ttyS2) and thus it can't be used. See Serial Port Devices /dev/ttyS2, etc. for more details>

3.4 Interrupts

When the serial port receives a number of bytes (may be set to 1, 4, 8, or 14) into its FIFO buffer, it signals the CPU to fetch them by sending an electrical signal known as an interrupt on a certain wire normally used only by that port. Thus the FIFO waits for a number of bytes and then issues an interrupt.

However, this interrupt will also be sent if there is an unexpected delay while waiting for the next byte to arrive (known as a timeout). Thus if the bytes are being received slowly (such as someone typing on a terminal keyboard) there may be an interrupt issued for every byte received. For some UART chips the rule is like this: If 4 bytes in a row could have been received, but none of these 4 show up, then the port gives up waiting for more bytes and issues an interrupt to fetch the bytes currently in the FIFO. Of course, if the FIFO is empty, no interrupt will be issued.

Each interrupt conductor (inside the computer) has a number (IRQ) and the serial port must know which conductor to use to signal on. For example, ttyS0 normally uses IRQ number 4 known as IRQ4 (or IRQ 4). A list of them and more will be found in "man setserial" (search for "Configuring Serial Ports"). Interrupts are issued whenever the serial port needs to get the CPU's attention. It's important to do this in a timely manner since the buffer inside the serial port can hold only 16 (1 in old serial ports) incoming bytes. If the CPU fails to remove such received bytes promptly, then there will not be any space left for any more incoming bytes and the small buffer may overflow (overrun) resulting in a loss of data bytes. There is no Flow Control to prevent this.

Interrupts are also issued when the serial port has just sent out all 16 of its bytes from its small transmit buffer out the external cable. It then has space for 16 more outgoing bytes. The interrupt is to notify the CPU of that fact so that it may put more bytes in the small transmit buffer to be transmitted. Also, when a modem control line changes state an interrupt is issued.

The buffers mentioned above are all hardware buffers. The serial port also has large buffers in main memory. This will be explained later

Interrupts convey a lot of information but only indirectly. The interrupt itself just tells a chip called the interrupt controller that a certain serial port needs attention. The interrupt controller then signals the CPU. The CPU runs a special program to service the serial port. That program is called an interrupt service routine (part of the serial driver software). It tries to find out what has happened at the serial port and then deals with the problem such a transferring bytes from (or to) the serial port's hardware buffer. This program can easily find out what has happened since the serial port has registers at IO addresses known to the the serial driver software. These registers contain status information about the serial port. The software reads these registers and by inspecting the contents, finds out what has happened and takes appropriate action.

3.5 Data Flow (Speeds)

Data (bytes representing letters, pictures, etc.) flows into and out of your serial port. Flow rates (such as 56k (56000) bits/sec) are (incorrectly) called "speed". But almost everyone says "speed" instead of "flow rate".

It's important to understand that the average speed is often less than the specified speed. Waits (or idle time) result in a lower average speed. These waits may include long waits of perhaps a second due to Flow Control. At the other extreme there may be very short waits (idle time) of several micro−seconds between bytes. If the device on the serial port (such as a modem) can't accept the full serial port speed, then the average speed must be reduced.

3.6 Flow Control

Flow control means the ability to slow done the flow of bytes in a wire. For serial ports this means the ability to stop and then restart the flow without any loss of bytes. Flow control is needed for modems to allow a jump in instantaneous flow rates.

Example of Flow Control

For example, consider the case where you connect a 36.6k external modem via a short cable to your serial port. The modem sends and receives bytes over the phone line at 36.6k bits per second (bps). Assume it's not doing any data compression or error correction. You have set the serial port speed to 115,200 bits/sec (bps), and you are sending data from your computer to the phone line. Then the flow from the your computer to your modem over the short cable is at 115.2k bps. However the flow from your modem out the phone line is only 33.6k bps. Since a faster flow (115.2k) is going into your modem than is coming out of it, the modem is storing the excess flow (115.2k −33.6k = 81.6k bps) in one of its buffers. This buffer would soon overrun (run out of free storage space) unless the high 115.2k flow is stopped.

But now flow control comes to the rescue. When the modem's buffer is almost full, the modem sends a stop signal to the serial port. The serial port passes on the stop signal on to the device driver and the 115.2k bps flow is halted. Then the modem continues to send out data at 33.6k bps drawing on the data it previous accumulated in its buffer. Since nothing is coming into the buffer, the level of bytes in it starts to drop. When almost no bytes are left in the buffer, the modem sends a start signal to the serial port and the 115.2k flow from the computer to the modem resumes. In effect, flow control creates an average flow rate in the short cable (in this case 33.6k) which is significantly less than the "on" flow rate of 115.2k bps. This is "start−stop" flow control.

In the above simple example it was assumed that the modem did no data compression. This would be true when the modem is sending a file which is already compressed and can't be compressed further. Now let's consider the opposite extreme where the modem is compressing the data with a high compression ratio. In such a case the modem might need an input flow rate of say 115.2k bps to provide an output (to the phone line) of 33.6k bps (compressed data). The compression ratio is 3.43 (115.2/33.6) which is much higher than average. In this case the modem is able to compress and the 115.2 bps PC−to−modem flow and send the same data out on the phone line at 33.6bps. There's no need for flow control here. But such a high compression ratio rarely happens so that most of the time flow control is needed to slow down the flow on the 115.2 bps PC−to−modem cable. The flow is stopped and started so that the average flow is usually well under the "on" flow of 115.2 bps.

In the above example the modem was an external modem. But the same situation exists (as of late 2000) for most internal modems. There is still a speed limit on the PC−to−modem speed even though this flow doesn't take place over an external cable. This makes the internal modems compatible with the external modems.

In the above example of flow control the flow was from the computer to a modem. But there is also flow control which is used for the opposite direction of flow: from a modem (or other device) to a computer. Each direction of flow involve 3 buffers: 1. in the modem 2. in the UART chip (called FIFOs) 3. in main memory managed by the serial driver. Flow control protects certain buffers from overflowing. The small UART FIFO buffers are not protected in this way but rely instead on a fast response to the interrupts they issue. FIFO stand for "First In, First Out" which is the way it handles bytes. All the 3 buffers use the FIFO rule but only one of them also uses it as a name. This is the essence of flow control but there are still some more details.

Symptoms of No Flow Control

Understanding flow−control theory can be of practical use. The symptom of no flow control is chunks of data missing from files sent without the benefit of flow control. This is because when overflow happens, it's usually more than just a few bytes that overflow and are lost. Often hundreds or even thousands of bytes get lost, and all in contiguous chunks.

Hardware vs. Software Flow Control

If feasible it's best to use "hardware" flow control that uses two dedicated "modem control" wires to send the "stop" and "start" signals.

Software flow control uses the main receive and transmit wires to send the start and stop signals. It uses the ASCII control characters DC1 (start) and DC3 (stop) for this purpose. They are just inserted into the regular stream of data. Software flow control is not only slower in reacting but also does not allow the sending of binary data unless special precautions are taken. Since binary data will likely contain DC1 and DC3, special means must be taken to distinguish between a DC3 that means a flow control stop and a DC3 that is part of the binary code. Likewise for DC1.

3.7 Data Flow Path; Buffers

Much has been explained about this including flow control, a pair of 16−byte FIFO buffers (in the UART), and a pair of larger buffers inside a device connected to the serial port (such as a modem. But there is still another pair of buffers. These are large buffers (perhaps 8k) in main memory also known as serial port buffers. When an application program sends bytes to the serial port

they first get stashed in the the transmit serial port buffer in main memory. The pair consists of both this transmit buffer and a receive buffer for the opposite direction of byte−flow. Here's an example diagram for the case of browsing the Internet with a browser. Transmit data flow is left to right while receive flow is right to left.

```
application     8k-byte         16-byte        1k-byte        tele-
BROWSER ------- MEMORY -------- UART --------- MODEM -------- phone
program         buffer          buffer         buffer         line
```

The serial device driver takes out say 16 bytes from this transmit buffer, one byte at a time and puts them into the 16−byte transmit buffer in the serial UART for transmission. Once in that transmit buffer, there is no way to stop them from being transmitted. They are then transmitted to the modem or other device connected to the serial port which also has a fair sized (say 1k) buffer. When the device driver (on orders from flow control) stops the flow of outgoing bytes from the computer, what it actually stops is the flow of outgoing bytes from the large transmit buffer in main memory. Even after this has happened and the flow to the device connected to the serial port has stopped, an application program may keep sending bytes to the 8k transmit buffer until it becomes fill.

When it gets fill, the application program can't send any more bytes to it (a "write" statement in a C_program blocks) and the application program temporarily stops running and waits until some buffer space becomes available. Thus a flow control "stop" is ultimately able to stop the program that is sending the bytes. Even though this program stops, the computer does not necessarily stop computing. It may switch to running other processes while it's waiting at a flow control stop. The above was a little oversimplified since there is another alternative of having the application program itself do something else while it is waiting to "write".

3.8 Complex Flow Control Example

For many situations, there is a transmit path involving several links, each with its own flow control. For example, I type at a text−terminal connected to a PC with a modem to access a BBS. For this I use the application program "minicom" which deals with 2 serial ports: one connected to a modem and another connected to the text−terminal. What I type at the text terminal goes into the first serial port to minicom, then from minicom out the second serial port to the modem, and then onto the telephone line to the BBS. The text−terminal has a limit to the speed at which bytes can be displayed on its screen and issues a flow control "stop" from time to time to slow down the flow. What happens when such a "stop" is issued? Let's consider a case where the "stop" is long enough to get thru to the BBS and stop the program at the BBS which is sending out the bytes.

Let's trace out the flow of this "stop" (which may be "hardware" on some links and "software" on others). First, suppose I'm "capturing" a long file from the BBS which is being sent simultaneously to both my text−terminal and a to file on my hard−disk. The bytes are coming in faster than the terminal can handle them so it sends a "stop" out its serial port to the first serial port on my PC. The device driver detects it and stops sending bytes from the 8k serial buffer (in main memory) to the terminal. Now minicom still keeps sending out bytes for the terminal into this 8k buffer.

When this 8k transmit buffer (on the first serial port) is full, minicom must stop writing to it. Minicom stops and waits. But this also causes minicom to stop reading from the 8k receive buffer on the 2nd serial port connected to the modem. Flow from the modem continues until this 8k buffer too fills up and sends a different "stop" to the modem. Now the modem's buffer ceases to send to the serial port and also fills up. The modem (assuming error correction is enabled) sends a "stop signal" to the other modem at the BBS. This modem stops sending bytes out of its buffer and when its buffer gets fill, another stop signal is sent to the serial port of the BBS. At the BBS, the 8−k (or whatever) buffer fills up and the program at the BBS can't write to it anymore and thus temporarily halts.

Thus a stop signal from a text terminal has halted a programs on a BBS computer. What a complex sequence of events! Note that the stop signal passed thru 4 serial ports, 2 modems, and one application program (minicom). Each serial port has 2 buffers (in one direction of flow): the 8k one and the hardware 16−byte one. The application program may have a buffer in its C_code. This adds up to 11 different buffers the data is passing thru. Note that the small serial hardware buffers do not participate directly in flow control.

If the terminal speed limitation is the bottleneck in the flow from the BBS to the terminal, then its flow control "stop" is actually stopping the program that is sending from the BBS as explained above. But you may ask: How can a "stop" last so long that 11 buffers (some of them large) all get filled up? It can actually happen this way if all the buffers were near their upper limits when the terminal sent out the "stop".

But if you were to run a simulation on it you would discover that it's usually more complicated than this. At an instant of time some links are flowing and others are stopped (due to flow control). A "stop" from the terminal seldom propagates back to the BBS neatly as described above. It may take a few "stops" from the terminal to result in one "stop" at the BBS. To understand what is going on you really need to observe a simulation which can be done for a simple case with coins on a table. Use only a few buffers and set the upper level for each buffer at only a few coins.

Does one really need to understand all this? Well, understanding this explained to me why capturing text from a BBS was loosing text. The situation was exactly the above example but modem−to−modem flow control was disabled. Chunks of captured text that were supposed to also get to my hard−disk never got there because of an overflow at my modem buffer due to flow control "stops" from the terminal. Even though the

BBS had a flow path to the hard−disk without bottlenecks, the overflow due to the terminal happened on this path and chunks of text were lost and never even made it to the hard−disk. Note that the flow to the hard−disk passed thru my modem and since the overflow happened there, bytes intended for the hard−disk were lost.

3.9 Serial Driver Module

The device driver for the serial port is the software that operates the serial port. It is now provided as a serial module. >From kernel 2.2 on, this module will normally get loaded automatically if it's needed. In earlier kernels, you had to have `kerneld` running in order to do auto−load modules on demand. Otherwise the serial module needed to be explicitly listed in /etc/modules. Before modules became popular with Linux, the serial driver was usually built into the kernel (and sometimes still is). If it's built−in don't let the serial module load or else you will have two serial drivers running at the same time. With 2 drivers there are all sorts of errors including a possible "I/O error" when attempting to open a serial port. Use "lsmod" to see if the module is loaded.

When the serial module is loaded it displays a message on the screen about the existing serial ports (often showing a wrong IRQ). But once the module is used by `setserial` to tell the device driver the (hopefully) correct IRQ then you should see a second display similar to the first but with the correct IRQ, etc. See Serial Module See What is Setserial for more info on `setserial`.

---

4. Is the Serial Port Obsolete?

4.1 Introduction

The answer is yes, but ... The serial port is somewhat obsolete but it's still needed, especially for Linux. The serial port has many shortcomings but almost all new PC's seem to come with them them. Linux supports ordinary telephone modems only if they work thru a serial port.

The serial port must pass data between the computer and the external cable. Thus it has two interfaces and both of these interfaces are slow. First we'll consider the interface via external cable to the outside world.

4.2 EIA−232 Cable Is Low Speed & Short Distance

The conventional EIA−232 serial port is inherently low speed and is severely limited in distance. Ads often read "high speed" but it can only work at "high speed" over very short distances such as to a modem located right next to the computer. Compared to a network card, even this "high speed" is low speed. All of the serial cable wires use a common ground return wire so that twisted−pair technology (needed for high speeds) can't be used without additional hardware. More modern interfaces for serial ports exist but they are not standard on PC's like the EIA−232 is. See Successors to EIA−232. Some multiport serial cards support them.

It is somewhat tragic that the RS−232 standard from 1969 did not use twisted pair technology which could operate about a hundred times faster. Twisted pairs have been used in telephone cables since the late 1800's. In 1888 (over 110 years ago) the "Cable Conference" reported its support of twisted−pair (for telephone systems) and pointed out its advantages. But over 80 years after this approval by the "Cable Conference", RS−232 failed to utilize it. Since RS−232 was originally designed for connecting a terminal to a low speed modem located nearby, the need for high speed and longer distance transmission was apparently not recognized.

4.3 Inefficient Interface to the Computer

To communicate with the computer, any I/O device needs to have an address so that the computer can write to it and read from it. For this purpose many I/O devices (such as serial ports) use a special type of address known as an I/O addresses (sometimes called an I/O port). It's actually a range of addresses and the lower address in this range is the base address. If someone only says (or writes) "address" it likely really means "base address"

Instead of using I/O, addresses some I/O devices read and write directly from/to main memory. This provides more bandwidth since the conventional serial I/O system only moves a byte at a time. There are various ways to read/write directly to main memory. One way is called shared memory I/O (where the shared memory is usually on the same card as the I/O device). Other methods are DMA (direct memory access) on the ISA bus and what is about the same as DMA (only much faster): "bus mastering" on the PCI bus. These methods are a lot faster than those used for the serial port. Thus the conventional serial port with its interrupt driven (every 14 bytes) interface and single bytes transfers on a bus which could accommodate 4 (or 8) bytes at a time is not suited for very high speed I/O.

---

5. Multiport Serial Boards/Cards/Adapters

5.1 Intro to Multiport Serial

Multiport serial cards install in slots in a PC on the ISA or PCI bus. Instead of being called "... cards" they are also called "... adapters" or "... boards". Each such card provides you with many serial ports. Today they are commonly used for the control of external devices (including automation for both industry and the home). They can connect to computer servers for the purpose of monitoring/controlling the server from a remote location. They were once mainly used for connecting up many terminals and/or modems to serial ports. They are still used this way but any modem used with it has the same limitation of ordinary modems: It can't send at over 33.6k even if it is a 56k modem.

Thus if someone dials in to you (reaches your multiport serial card from a modem plugged into the card) they will not be able to go above 33.6k in either direction, even if they use a 56k modem. To go above 33.6k for dial−in requires that you have a digital connection to the telephone line. The serial port can't be used for this case. Thus serial multiport cards are now obsolete for use by ISPs or anyone that needs to allow others to dial−in to them at 56k (over 33.6k). See Modem−HOWTO: Modem Pools, Digital Modems.

Each multiport card has a number of external connecters (DB−25 or RJ45) so that one may connect up a number of devices (modems, terminals, etc.). Each such physical device would then be connected to its own serial port. Since the space on the external−facing part of the card is limited there is often not enough room for all the serial port connectors. To solve this problem, the connectors may be on the ends of cables which come out (externally) from the card (octopus cable). Or they may be on an external box (possibly rack mountable) which is connected by a cable to a multiport card.

Dumb multiport cards are not too much different than ordinary serial ports. They are interrupt driven and the CPU of the computer does most all the work servicing them. They usually have a system of sharing a single interrupt for all the ports. This doesn't decrease the load on the CPU since the single interrupt will be sent to the CPU each time any of the ports needs servicing. Such devices usually require special drivers that you must put into the kernel or activate by putting a #define in the source code (or the like).

Smart boards may use ordinary UARTs but handle most interrupts from the UARTs internally within the board. This frees the CPU from the burden of handling all these interrupts. The board may save up bytes in its large internal FIFOs and transfer perhaps 1k bytes at a time to the serial buffer in main memory. It may use the full bus width of 32 bits for making data transfers to main memory (instead of transferring only 8−bit bytes like dumb serial cards do). Not all "smart" boards are equally efficient. Many boards today are Plug−and−Play.

For a smart board to work, a special driver for it must be used. Sometimes this driver is built into the kernel source code or supplied as a module. Even in such cases, you may need to do something to activate it. This includes selecting it when you compile the kernel (during the configuration phase just before you compile). A pre−compiled kernel probably didn't put the smart board driver into the kernel but it may come with a pre−compiled module for the board. This module needs to be loaded but the kernel may automatically do this for you if a program is trying to use a device on the smart board (provided there exists a table showing which module to load for the device). This table may be in /etc/modules.conf and/or be internal to the kernel. Also certain parameters may need to be passed to the driver (via lilo's append command or via /etc/modules.conf).

The board's manufacturer should have info on all this on their website. Unfortunately, info for old boards is sometimes not there but might be found somewhere else on the Internet (including discussion groups). You might also want to look at the kernel documentation in /usr/share/kernel−doc... For configuring the kernel or modules prior to compiling see: Configure.help and search for "serial", etc. There are also kernel documentation files for certain boards including computone, hayes−esp, moxa−smartio, riscom8, specialix, stallion, and sx.

5.2 Making multiport devices in the /dev directory

The serial ports your multiport board uses depends on what kind of board you have. Some have their own device names like /dev/ttyE27 or /dev/ttyD2, etc. Ones that use the standard names like /dev/ttyS14 may be listed in detail in `rc.serial` or in `0setserial`. These files may be included in a setserial or serial package. You may need to create these devices (unless an installation script does it for you). Either use the `mknod` command, or the `MAKEDEV` script. Devices (in the /dev directory) for ttyS type serial ports are made by adding ``64 + port number''. So, if you wanted to create devices for `ttyS17`, you would type:

```
linux# mknod −m 666 /dev/ttyS17 c 4 81
```

Note the "major" number is always 4 for ttyS devices (and 5 for the obsolete cua devices). Also ``64 + 17 = 81''. Using the `MAKEDEV` script, you would type:

```
linux# cd /dev
linux# ./MAKEDEV ttyS17
```

For the names and numbers of other types of serial ports other than ttyS.. see devices.txt in the kernel documentation. Besides the listing of various brands of multiports found in this HOWTO there is Gary's Encyclopedia − Serial Cards. It's not as complete, but may have some different links.

5.3 Standard PC Serial Cards

In olden days PCs came with a serial card installed. Later on the serial function was put on the hard−drive interface card. Today, one or two serial ports are usually built into the motherboard. Most of them (as of 2001) use a 16550 but some use 16650 (32−byte FIFOs). But one may still buy the old PC serial cards if they need 1−4 more serial ports. These are for ttyS0−ttyS3 (COM1 − COM4). They can be used to connect external serial devices (modems, serial mice, etc...). Only a tiny percentage of retail computer stores carry such cards. But one can purchase them on the Internet. Before getting a PCI one, make sure Linux supports it.

Here's a list of a few popular brands:

- Byte Runner (may order directly, shows prices) http://www.byterunner.com
- SIIG http://www.siig.com/io
- Dolphin http://www.dolphinfast.com/sersol/

Note: due to address conflicts, you may not be able to use COM4 and IBM8514 video card (or some others) simultaneously. See Avoiding IO Address Conflicts with Certain Video Boards

5.4 Dumb Multiport Serial Boards (with standard UART chips)

They are also called "serial adapters". They often have a special method of sharing interrupts which requires that you compile support for them into the kernel.

* => The file that ran setserial in Debian shows some details of configuring # => See note below for this board

- AST FourPort and clones (4 ports) * #
- Accent Async−4 (4 ports) *
- Arnet Multiport−8 (8 ports)
- Bell Technologies HUB6 (6 ports)
- Boca BB−1004 (4 ports), BB−1008 (8 ports), BB−2016 (16 ports; See the Boca mini−howto revised in 2001) * #
- Boca IOAT66 or? ATIO66 (6 ports, Linux doesn't support its IRQ sharing ?? Uses odd−ball 10−cond RJ45−like connectors)
- Boca 2by4 (4 serial ports, 2 parallel ports)
- Byte Runner http://www.byterunner.com
- Computone ValuePort V4−ISA (AST FourPort compatible) *
- Digi PC/8 (8 ports) #
- Dolphin http://www.dolphinfast.com/sersol/
- Globetek http://www.globetek.com/
- GTEK BBS−550 (8 ports; See the mini−howto)
- Hayes ESP (after kernel 2.1.15)
- HUB−6 See Bell Technologies.
- Longshine LCS−8880, Longshine LCS−8880+ (AST FourPort compatible) *
- Moxa C104, Moxa C104+ (AST FourPort compatible) *
- NI−SERIAL by National Instruments
- PC−COMM (4 ports)
- Sealevel Systems COMM−2 (2 ports), COMM−4 (4 ports) and COMM−8 (8 ports)
- SIIG I/O Expander 2S IO1812 (4 ports) #
- STB−4COM (4 ports)
- Twincom ACI/550
- Usenet Serial Board II (4 ports) *
- VScom (uses same driver as ByteRunner)

In general, Linux will support any serial board which uses a 8250, 16450, 16550, 16550A, 16650, etc. UART. See the latest man page for "setserial" for a more complete list.

Notes:

AST Fourport: You might need to specify `skip_test` in `rc.serial`.

BB−1004 and BB−1008 do not support DCD and RI lines, and thus are not usable for dialin modems. They will work fine for all other purposes.

Digi PC/8 Interrupt Status Register is at 0x140.

SIIG IO1812 manual for the listing for COM5−COM8 is wrong. They should be COM5=0x250, COM6=0x258, COM7=0x260, and COM8=0x268.

5.5 Intelligent Multiport Serial Boards

Make sure that a Linux−compatible driver is available and read the information that comes with it. These boards use special devices (in the /dev directory), and not the standard ones. This information varies depending on your hardware. If you have updated info which should be shown here please email it to me.

Names of Linux driver modules are *.o but these may not work for all models shown. Also, parameters (such as the io and irq often need to be given to the module so you need to find instructions on this (possibly in the source code tree). To check on the latest serial driver go to Linux Serial Driver home page

There are many different brands, each of which often offers many different cards. No attempt is currently being made to list all the cards here (and many listed are obsolete). But all major brands and websites should be shown here so it something is missing let me know. Go the the webpage shown for more information. These websites often also have info (ads) on related hardware such as modem pools, remote access servers (RASs), and terminal servers. Where there is no webpage, the cards are likely obsolete. If you would like to put together a better list, let me know.

- Chase Research (UK based, ISA/PCI cards)
  webpage: `www.chaser.com`
  driver status: for 2.2 kernel. Supported by Chase.
- Comtrol RocketPort (36MHz ASIC; 4, 8, 16, 32, up to 128 ports)
  webpage: `http://www.comtrol.com`
  driver status: supported by Comtrol. rocket.o
  driver location: `ftp://tsx-11.mit.edu/pub/linux/packages/comtrol`
- Computone IntelliPort II (ISA, PCI and EISA busses up to 64 ports)
  webpage: http://www.computone.com
  driver location: ftp://ftp.computone.com/PUB/Products/IntelliPortII/Linux/, patch at http://www.wittsend.com/computone/linux−2.2.10−ctone.patch.gz
  mailing list: mailto:majordomo@lazuli.wittsend.com with "subscribe linux−computone" in body
  note: Old ATvantage and Intelliport cards are not supported by Computone
- Connecttech
  website: `http://www.connecttech.com/porducts/products.html`
  driver location: ftp://ftp.connecttech.com/pub/linux/
- Cyclades
  Cyclom−Y (Cirrus Logic CD1400 UARTs; 8 − 32 ports),

Cyclom−Z (MIPS R3000; 8 − 64 ports)
website: `http://www.cyclades.com/products.html`
driver status: supported by Cyclades
driver location: `ftp://ftp.cyclades.com/pub/cyclades` and included in Linux kernel since version 1.1.75: cyclades.o

- Decision PCCOM (2−8 ports; ISA and PCI; AKA PC COM)
  ISA:
  contact: `mailto:info@cendio.se`
  driver location: `ftp://ftp.signum.se/pub/pccom8`
  PCI:
  drivers: http://www.decision.com.tw Click "download"
  driver status: Support in serial driver 5.03. For an earlier driver, there exists a patch for kernel 2.2.16 at http://www.qualica.com/serial/
- Digi PC/Xi (12.5MHz 80186; 4, 8, or 16 ports),
  PC/Xe (12.5/16MHz 80186; 2, 4, or 8 ports),
  PC/Xr (16MHz IDT3041; 4 or 8 ports),
  PC/Xem (20MHz IDT3051; 8 − 64 ports)
  website: `http://www.dgii.com`
  driver status: supported by Digi
  driver location: `ftp://ftp.dgii.com/drivers/linux` and included in Linux kernel since version 2.0. epca.o
- Digi COM/Xi (10MHz 80188; 4 or 8 ports)
  contact: Simon Park, `si@wimpol.demon.co.uk`
  driver status: ?
  note: Simon is often away from email for months at a time due to his job. Mark Hatle, mailto:fray@krypton.mankato.msus.edu has graciously volunteered to make the driver available if you need it. Mark is not maintaining or supporting the driver.
- Equinox SuperSerial Technology (30MHz ASIC; 2 − 128 ports)
  website: `http://www.equinox.com`
  driver status: supported by Equinox
  driver location: `ftp://ftp.equinox.com/library/sst`
- Globetek
  website: http://www.globetek.com/products.shtml
  driver location: http://www.globetek.com/media/files/linux.tar.gz
- GTEK Cyclone (16C654 UARTs; 6, 16 and 32 ports),
  SmartCard (24MHz Dallas DS80C320; 8 ports),
  BlackBoard−8A (16C654 UARTs; 8 ports),
  PCSS (15/24MHz 8032; 8 ports)
  website: `http://www.gtek.com`
  driver status: supported by GTEK
  driver location: `ftp://ftp.gtek.com/pub`
- Hayes ESP (COM−bic; 1 − 8 ports)
  website: `http://www.nyx.net/~arobinso`
  driver status: Supported by Linux kernel (1998) since v. 2.1.15. esp.o. Setserial 2.15+ supports. Also supported by author
  driver location: `http://www.nyx.net/~arobinso`
- Intelligent Serial Interface by Multi−Tech Systems
  PCI: 4 or 8 port. ISA 8 port. DTE speed 460.8k
  webpage: http://www.multitech.com/products/
- Maxpeed SS (Toshiba; 4, 8 and 16 ports)
  website: `http://www.maxpeed.com`

driver status: supported by Maxpeed
driver location: `ftp://maxpeed.com/pub/ss`

- Microgate SyncLink ISA and PCI high speed multiprotocol serial. Intended for synchronous HDLC.
  website: http://ww/microgate.com/products/sllinux/hdlcapi.htm
  driver status: supported by Microgate: synclink.o
- Moxa C218 (12MHz 80286; 8 ports),
  Moxa C320 (40MHz TMS320; 8 – 32 ports)
  website: `http://www.moxa.com`
  driver status: supported by Moxa
  driver locations:
  `http://www.moxa.com/support/download/download.php3>` `ftp://ftp.moxa.com/drivers/linux` (also from Taiwan at www.moxa.com.tw/...) where ... is the same as above)
- SDL RISCom/8 (Cirrus Logic CD180; 8 ports)
  website: `http://www.sdlcomm.com`
  driver status: supported by SDL
  driver location: `ftp://ftp.sdlcomm.com/pub/drivers`
- Specialix SX (25MHz T225; 8? – 32 ports),
  SIO/XIO (20 MHz Zilog Z280; 4 – 32 ports)
  webpage: www.specialix.com/products/io/serialio.htm
  driver status: Supported by Specialix
  driver location: http://www.BitWizard.nl/specialix/
  old driver location: ftp://metalab.unc.edu/pub/Linux/kernel/patches/serial
- Stallion EasyIO–4 (4 ports), EasyIO–8 (8 ports), and
  EasyConnection (8 – 32 ports) – each with Cirrus Logic CD1400 UARTs,
  Stallion (8MHz 80186 CPU; 8 or 16 ports),
  Brumby (10/12 MHz 80186 CPU; 4, 8 or 16 ports),
  ONboard (16MHz 80186 CPU; 4, 8, 12, 16 or 32 ports),
  EasyConnection 8/64 (25MHz 80186 CPU; 8 – 64 ports)
  contact: `sales@stallion.com` or `http://www.stallion.com`
  driver status: supported by Stallion
  driver location: `ftp://ftp.stallion.com/drivers/ata5/Linux` and included in linux kernel since 1.3.27
- System Base website: http://www.sysbas.com/

A review of Comtrol, Cyclades, Digi, and Stallion products was printed in the June 1995 issue of the *Linux Journal*. The article is available at `http://www.ssc.com/lj/issue14`.

5.6 Unsupported Multiport Boards

The following boards don't mention any Linux support as of 1 Jan. 2000. Let me know if this changes.

- Aurora (PCI only) www.auroratech.com

---

6. Configuring the Serial Port

6.1 PCI Bus Support Underway

Although most PCI modems are "winmodems" without a Linux driver (and will not work under Linux), other PCI serial cards (usually modem cards) will often work OK under Linux. Some need no special support in the serial driver and work fine under Linux once setserial is used to configure them. Other PCI cards need special support in the kernel. Some of these cards are supported by kernel 2.4 (and in later versions of the 2.3 series). Kernel 2.2 has no such support. If your modem (or serial port) happens to be supported, then you shouldn't need to do anything to PnP configure it. The new serial driver will read the id number digitally stored on the card to determine how to support the card. It should assign the I/O address to it, determine it's IRQ, etc. So you don't need to use "setserial" for it.

If you have a

PCI modem card you should be looking at Modem−HOWTO and not this Serial−HOWTO. If you just have a PCI serial port card (with no modem on the card) but it will not work because the latest serial driver doesn't support it, you can help in attempting to create a driver for it. To do this you'll need to contact the maintainer of the serial driver, Theodore (Ted) Y. Ts'o. You will need to email Ted Ts'o a copy of the output of "lspci −vv" with full information about the model and manufacturer of the PCI modem (or serial port). Then he will try to point you to a test driver which might work for it. You will then need to get it, compile it and possibly recompile your kernel. Then you will test the driver to see if it works OK for you and report the results to Ted Ts'o. If you are willing to do all the above (and this is the latest version of this HOWTO) then email the needed info to him at: mailto:tytso@mit.edu.

PCI modems are not well standardized. Some use main memory for communication with the PC. It you see 8−digit hexadecimal addresses it's not likely to work with Linux. Some require special enabling of the IRQ. The output of "lspci" can help determine if one can be supported. If you see a 4−digit IO port and no long memory address, the modem might work by just telling "setserial" the IO port and the IRQ. Some people have gotten a 3COM 3CP5610 PCI Modem to work that way.

6.2 Configuring Overview

In many cases, configuring will happen automatically and you have nothing to do. But sometimes you need to configure (or just want to check out the configuration). If so, first you need to know about the two parts to configuring the serial port under Linux:

The first part (low−level configuring) is assigning it an IO address, IRQ, and name (such as ttyS2). This IO−IRQ pair must be set in both the hardware and told to the serial driver. We might just call this "io−irq" configuring for short. The `setserial` is used to tell the driver. PnP methods, jumpers, etc, are used to set the hardware. Details will be supplied later. If you need to configure but don't understand certain details it's easy to get into trouble.

The second part (high−level configuring) is assigning it a speed (such as 38.4k bits/sec), selecting flow control, etc. This is often done by communication programs such as PPP, minicom, or by getty (which you may run on the port so that others may log into your computer). However you will need to tell these programs what speed you want, etc. by using a menu or a configuration file. This high−level configuring may also be done with the `stty` program. `stty` is also useful to view the current status if you're having problems. See also the section Stty When Linux starts, some effort is made to detect and configure (low−level) a few serial ports. Exactly what happens depends on your BIOS, hardware, Linux distribution, etc. If the serial ports work OK, there may be no need for you to do any configuring. Application programs often do the high−level configuring but you may need to supply them with the required information. With Plug−and−Play serial ports (often built into an internal modem), the situation has become more complex. Here are cases when you need to do low−level configuring (set IRQ and IO addresses):

- Plan to use more than 2 serial ports
- Installing a new serial port (such as an internal modem)
- Having problems with serial port(s)

For kernel 2.2+ you may be able to use more that 2 serial ports without low−level configuring by sharing interrupts. This only works if the serial hardware supports it and may be no easier than low−level configuring. See Interrupt sharing and Kernels 2.2+

The low−level configuring (setting the IRQ and IO address) seems to cause people more trouble (than high−level), although for many it's fully automatic and there is no configuring to be done. Thus most all of this section is on that topic. Until the serial driver knows the correct IRQ and IO address the port will not work at all. It may not even be found by Linux. Even if it can be found, it may work extremely slow if the IRQ is wrong. See Extremely Slow: Text appears on the screen slowly after long delays.

In the Wintel world, the IO address and IRQ are called "resources" and we are thus configuring certain resources. But there are many other types of "resources" so the term has many other meanings. In review, the low−level configuring consists of putting two values (an IRQ number and IO address) into two places:

1. the device driver (often by running "`setserial`" at boot−time)
2. memory registers of the serial port hardware itself

You may watch the start−up (= boot−time) messages. They are usually correct. But if you're having problems, there's a good chance that some of these messages don't show the true configuration of the hardware (and they are not supposed to). See I/O Address & IRQ: Boot−time messages.

6.3 Common mistakes made re low−level configuring

Here are some common mistakes people make:

- setserial command: They run it (without the "autoconfig" and auto_irq options) and think it has checked out the hardware (it hasn't).
- setserial messages: They see them displayed on the screen at boot−time (or by giving the setserial command) and erroneously think that the result always shows how their hardware is actually configured.
- /proc/interrupts: When their serial device isn't in use they don't see its interrupt there, and erroneously conclude that their serial port can't be found (or doesn't have an interrupt set).
- /proc/ioports and /proc/tty/driver/serial: People think this shows the actual hardware configuration when it only shows about the same info (possibly erroneous) as setserial.

6.4 I/O Address & IRQ: Boot−time messages

In many cases your ports will automatically get low−level configured at boot−time (but not always correctly). To see what is happening, look at the start−up messages on the screen. Don't neglect to check the messages from the BIOS before Linux is loaded (no examples shown here). These BIOS messages may be frozen by pressing the Pause key. Use Shift−PageUp to scroll back to the messages after they have flashed by. Shift−PageDown will scroll in the opposite direction. The `dmesg` command may be used at any time to view some of the messages but it often misses important ones. Here's an example of the start−up messages (as of mid 1999). Note that ttyS00 is the same as /dev/ttyS0.

```
At first you see what was detected (but the irq is only a wild guess):

Serial driver version 4.27 with no serial options enabled
ttyS00 at 0x03f8 (irq = 4) is a 16550A
ttyS01 at 0x02f8 (irq = 3) is a 16550A
ttyS02 at 0x03e8 (irq = 4) is a 16550A

Later you see what was saved, but it's not necessarily correct either:

Loading the saved−state of the serial devices...
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
/dev/ttyS2 at 0x03e8 (irq = 5) is a 16550A
```

Note that there is a slight disagreement: The first message shows ttyS2 at irq=4 while the second shows it at irq=5. Your may only have the first message. In most cases the last message is the correct one. But if your having trouble it may be misleading. Before reading the explanation of all of this complexity in the rest of this section, you might just try using your serial port and see if it works OK. If so it may not be essential to read further.

The second message is from the `setserial` program being run at boot−time. It shows what the device driver thinks is the correct configuration. But this too could be wrong. For example, the irq could actually be set to irq=8 in the hardware (both messages wrong). The irq=5 could be there because someone incorrectly put this into a configuration file (or the like). The fact that Linux sometimes gets IRQs wrong is because it doesn't by default probe for IRQs. It just assumes the "standard" ones (first message) or accepts what you told it when you configured it (second message). Neither of these is necessarily correct. If the serial driver has the wrong IRQ the serial port is very slow or doesn't seem to work at all.

The first message is a result of Linux probing the serial ports but it doesn't probe for IRQs. If a port shows up here it exists but the IRQ may be wrong. Linux doesn't check IRQs because doing so is not foolproof. It just assumes the IRQs are as shown because they are the "standard" values. Your may check them manually with `setserial` using the `autoconfig` and `auto_irq` options but this isn't guaranteed to be correct either.

The data shown by the BIOS messages (which you see at first) is what is set in the hardware. If your serial port is Plug−and−Play PnP then it's possible that the `isapnp` will run and change these settings. Look for messages about this after Linux starts. The last serial port message shown in the example above should agree with the BIOS messages (as possibly modified by isapnp). If they don't agree then you either need to change the setting in the port hardware or use setserial to tell the driver what is actually set in the hardware.

Also, if you have Plug−and−Play (PnP) serial ports, Linux will not find them unless the IRQ and IO has been set inside the hardware by Plug−and−Play software. Prior to kernel 2.4 this was a common reason why the start−up messages did not show a serial port that physically exists. The PC hardware (a PnP BIOS) may automatically low−level configure this. PnP configuring will be explained later.

6.5 What is the current IO address and IRQ of my Serial Port ?

If your serial port seems to work OK, then you may type "setserial −g /dev/ttyS*", look at /proc/tty/driver/serial, or inspect the start−up messages. If you serial port doesn't work (or is very slow) then you need to read further.

There are really two answers to the question "What is my IO and IRQ?" 1. What the device driver thinks has been set (This is what setserial usually sets and shows). 2. What is actually set in the hardware. They both should be the same. If they're not it spells trouble since the driver has incorrect info on the physical serial port. If the driver has the wrong IO address it will try to send data to a non−existing serial port −−or even worse, to some other device. If it has the wrong IRQ the driver will not get interrupt service requests from the serial port, resulting in a very slow or no response. See Extremely Slow: Text appears on the screen slowly after long delays. If it has the wrong model of UART there is also apt to be trouble. To determine if both I0−IRQ pairs are identical you must find out how they are set in both the driver and the hardware.

What does the device driver think?

This is easy to find out. Just look at the start−up messages or type "setserial −g /dev/ttyS*". If everything works OK then what it tells you is likely also set in the hardware. There are some other ways to find this info by looking at "files" in the /proc directory. Be warned that there is no guarantee that the same is set in the hardware.

`/proc/ioports` will show the IO addresses that the drivers are using. `/proc/interrupts` shows the IRQs that are used by drivers of currently running processes (that have devices open). It shows how many interrupts have actually be issued. `/proc/tty/driver/serial` shows most of the above, plus the number of bytes that have been received and sent (even if the device is not now open).

Note that for the IO addresses and IRQ assignments, you are only seeing what the driver thinks and not necessarily what is actually set in the hardware. The data on the actual number of interrupts issued and bytes processed is real however. If you see a large number of interrupts and/or bytes then it probably means that the device is (or was in the case of bytes) working. If there are no bytes received (rx:0) but bytes were transmitted (tx:3749 for example), then only one direction of flow is working (or being utilized).

Sometimes a showing of just a few interrupts doesn't mean that the interrupt is actually being physically generated by any serial port. Thus if you see almost no interrupts for a port that you're trying to use, that interrupt might not be set in the hardware and it implies that the driver is using the wrong interrupt. To view /proc/interrupts to check on a program that you're currently running (such as "minicom") you need to keep the program running while you view it.

What is set in my serial port hardware ?

How do you find out what IO address and IRQ are actually set in the device hardware? Perhaps the BIOS messages will tell you some info before Linux starts booting. Use the shift−PageUp key to step back thru the boot−time messages and look at the very first ones which are from the BIOS. This is how it was before Linux started. Setserial can't change it but isapnp or pciutils can and starting with kernel 2.4, these will be built into the serial driver.

One crude method is try probing with setserial using the "autoconfig" option. You'll need to guess the addresses to probe at. See What is Setserial. For a PCI serial port, use the "lspci" command (for kernels <2.2 look at /proc/pci). If your serial port is is Plug−and−Play see the next two subsections.

For a port set with jumpers, its how the jumpers were set. If the port is not Plug−and−Play (PnP) but has been setup by using a DOS program then it's set at whatever the person who ran that program set it to.

What is set in my PnP serial port hardware ?

PnP ports don't store their configuration in the hardware when the power is turned off. This is in contrast to Jumpers (non−PnP) which remain the same with the power off. If you have an ISA PnP port, it can reach a state where it doesn't have any IO address or IRQ and is in effect disabled. It should still be possible to find the port using the `pnpdump` program.

For Plug−and−Play (PnP) on the ISA bus one may try the `pnpdump` program (part of `isapnptools`). If you use the −−dumpregs option then it should tell you the actual IO address and IRQ set in the port. The address it "trys" is not the device's IO address, but a special

For PnP ports checking on how it's configured under DOS/Windows may not be of much help. Windows stores its configuration info in its Registry which is not used by Linux. It may supply the BIOS's non−volatile memory with some info but it may not be kept in sync with the current Window configuration in the Registry ?? If you let a PnP BIOS automatically do the configuring when you start Linux (and have told the BIOS that you don't have a PnP operating system when running Linux) then Linux should use whatever configuration is in the BIOS's non−volatile memory.

6.6 Choosing Serial IRQs

If you have Plug−and−Play ports then either a PnP BIOS or the serial driver may configure all your devices for you and then you may not need to choose any IRQs. PnP determines what it thinks is best and assigns them. But if you use the tools in Linux for Plug−and−Play (isapnp and pcitools) or jumpers then you have to choose. If you already know what IRQ you want to use you could skip this section except that you may want to know that IRQ 0 has a special use (see the following paragraph).

IRQ 0 is not an IRQ

While IRQ 0 is actually the timer (in hardware) it has a special meaning for setting a serial port with setserial. It tells the driver that there is no interrupt for the port and the driver then will use polling methods. This is quite inefficient but can be tried if there is an interrupt conflict or mis−set interrupt. The advantage of assigning this is that you don't need to know what interrupt is set in the hardware. It should be used only as a temporary expedient until you are able to find a real interrupt to use.

Interrupt sharing and Kernels 2.2+

The general rule is that every device should use a unique IRQ and not share them. But there are situations where sharing is permitted such as with most multi−port boards. Even when it is permitted, it may not be as efficient since every time a shared interrupt is given a check must be made to determine where it came from. Thus if it's feasible, it's nice to allocate every device its own interrupt.

Prior to kernel 2.2, serial IRQs could be shared with each other only for most multiport boards. Starting with kernel 2.2 serial IRQs may be sometimes shared between all serial ports. In order for sharing to work in 2.2 the kernel must have been compiled with CONFIG_SERIAL_SHARE_IRQ, and the serial port hardware must support sharing (so that if two serial cards put different voltages on the same interrupt wire, only the voltage that means "this is an interrupt" will prevail). Thus even if you have 2.2, it may be best to avoid sharing.

What IRQs to choose?

The serial hardware often has only a limited number of IRQs it can be set at. Also you don't want IRQ conflicts. So there may not be much of a choice. Your PC may normally come with `ttyS0` and `ttyS2` at IRQ 4, and `ttyS1` and `ttyS3` at IRQ 3. Looking at `/proc/interrupts` will show which IRQs are being used by programs currently running. You likely don't want to use one of these. Before IRQ 5 was used for sound cards, it was often used for a serial port.

Here is how Greg (original author of Serial−HOWTO) set his up in /etc/rc.d/rc.serial. rc.serial is a file (shell script) which runs at start−up (it may have a different name of location). For versions of "setserial" after 2.15 it's not always done this way anymore but this example does show the choice of IRQs.

```
/sbin/setserial /dev/ttyS0 irq 3        # my serial mouse
/sbin/setserial /dev/ttyS1 irq 4        # my Wyse dumb terminal
/sbin/setserial /dev/ttyS2 irq 5        # my Zoom modem
/sbin/setserial /dev/ttyS3 irq 9        # my USR modem
```

Standard IRQ assignments:

```
IRQ  0    Timer channel 0 (May mean "no interrupt".  See below.)
IRQ  1    Keyboard
IRQ  2    Cascade for controller 2
IRQ  3    Serial port 2
IRQ  4    Serial port 1
IRQ  5    Parallel port 2, Sound card
IRQ  6    Floppy diskette
IRQ  7    Parallel port 1
IRQ  8    Real−time clock
IRQ  9    Redirected to IRQ2
IRQ 10    not assigned
IRQ 11    not assigned
IRQ 12    not assigned
IRQ 13    Math coprocessor
IRQ 14    Hard disk controller 1
IRQ 15    Hard disk controller 2
```

There is really no Right Thing to do when choosing interrupts. Just make sure it isn't being used by the motherboard, or any other boards. 2, 3, 4, 5, 7, 10, 11, 12 or 15 are possible choices. Note that IRQ 2 is the same as IRQ 9. You can call it either 2 or 9, the serial driver is very understanding. If you have a very old serial board it may not be able to use IRQs 8 and above.

Make sure you don't use IRQs 1, 6, 8, 13 or 14! These are used by your motherboard. You will make her very unhappy by taking her IRQs. When you are done, double−check `/proc/interrupts` when programs that use interrupts are being run and make sure there are no conflicts.

6.7 Choosing Addresses −−Video card conflict with ttyS3

The IO address of the IBM 8514 video board (and others like it) is allegedly 0x?2e8 where ? is 2, 4, 8, or 9. This may conflict with the IO address of `ttyS3` at 0x02e8. Your may think that this shouldn't happen since the addresses are different in the high order digit (the leading 0 in 02e8). You're right, but a poorly designed serial port may ignore the high order digit and respond to any address that ends in 2e8. That is bad news if you try to use `ttyS3` (ISA bus) at this IO address.

For the ISA bus you should try to use the default addresses shown below. Addresses shown represent the first address of an 8−byte range. For example 3f8 is really the range 3f8−3ff. Each serial device (as well as other types of devices that use IO addresses) needs its own unique address range. There should be no overlaps (conflicts). Here are the default addresses for commonly used serial ports on the ISA bus:

```
ttyS0 address 0x3f8
ttyS1 address 0x2f8
ttyS2 address 0x3e8
ttyS3 address 0x2e8
```

Suppose there is an address conflict (as reported by `setserial −g /dev/ttyS*`) between a real serial port and another port which does not physically exist (and shows UART: unknown). Such a conflict shouldn't cause problems but it sometimes does in older kernels. To avoid this problem don't permit such address conflicts or delete /dev/ttySx if it doesn't physically exist.

6.8 Set IO Address & IRQ in the hardware (mostly for PnP)

After it's set in the hardware don't forget to insure that it also gets set in the driver by using `setserial`. For non−PnP serial ports they are either set in hardware by jumpers or by running a DOS program ("jumperless") to set them (it may disable PnP). The rest of this subsection is only for PnP serial ports. Here's a list of the possible methods of configuring PnP serial ports:

- Using a PnP BIOS CMOS setup menu (usually only for external devices on ttyS0 (Com1) and ttyS1 (Com2))
- Letting a PnP BIOS automatically configure a PnP serial port See Using a PnP BIOS to I0−IRQ Configure
- Doing nothing if you have both a PnP serial port and a PnP Linux operating system (see Plug−and−Play−HOWTO).
- Using `isapnp` for a PnP serial port non−PCI)
- Using pciutils (pcitools) for the PCI bus

The IO address and IRQ must be set (by PnP) in their registers each time the system is powered on since PnP hardware doesn't remember how it was set when the power is shut off. A simple way to do this is to let a PnP BIOS know that you don't have a PnP OS and the BIOS will automatically do this each time you start. This might cause problems in Windows (which is a PnP OS) if you start Windows with the BIOS thinking that Windows is not a PnP OS. See Plug−and−Play−HOWTO.

Plug−and−Play was designed to automate this io−irq configuring, but for Linux at present, it has made life more complicated. The standard kernels for Linux don't support plug−and−play very well. If you use a patch to the Linux kernel to covert it to a plug−and−play operating system, then all of the above should be handled automatically by the OS. But when you want to use this to automate configuring devices other that the serial port, you may find that you'll still have to configure the drivers manually since many Linux drivers are not written to support a Linux PnP OS. If you use `isapnptools` or the BIOS for configuring plug−and−play this will only put the two values into the registers of the serial port section of the modem card and you will likely still need to set up setserial. None of this is easy or very well documented as of early 1999. See Plug−and−Play−HOWTO and the isapnptools FAQ.

Using a PnP BIOS to I0−IRQ Configure

While the explanation of how to use a PnP OS or isapnp for io−irq configuring should come with such software, this is not the case if you want to let a PnP BIOS do such configuring. Not all PnP BIOS can do this. The BIOS usually has a CMOS menu for setting up the first two serial ports. This menu may be hard to find and for an "Award" BIOS it was found under "chipset features setup" There is often little to choose from. Unless otherwise indicated in a menu, these first two ports normally get set at the standard IO addresses and IRQs. See Serial Port Device Names & Numbers

Whether you like it or not, when you start up a PC a PnP BIOS starts to do PnP (io−irq) configuring of hardware devices. It may do the job partially and turn the rest over to a PnP OS (which you probably don't have) or if thinks you don't have a PnP OS it may fully configure all the PnP devices but not configure the device drivers. This is what you want but it's not always easy to figure out exactly what the PnP BIOS has done.

If you tell the BIOS that you don't have a PnP OS, then the PnP BIOS should do the configuring of all PnP serial ports −−not just the first two. An indirect way to control what the BIOS does (if you have Windows 9x on the same PC) is to "force" a configuration under Windows. See Plug−and−Play−HOWTO and search for "forced". It's easier to use the CMOS BIOS menu which may override what you "forced" under Windows. There could be a BIOS option that can set or disable this "override" capability.

If you add a new PnP device, the BIOS should change its PnP configuration to accommodate it. It could even change the io−irq of existing devices if required to avoid any conflicts. For this purpose, it keeps a list of non−PnP devices provided that you have told the BIOS how these non−PnP devices are io−irq configured. One way to tell the BIOS this is by running a program called ICU under DOS/Windows.

But how do you find out what the BIOS has done so that you set up the device drivers with this info? The BIOS itself may provide some info, either in its setup menus of via messages on the screen when you turn on your computer. See What is set in my serial port hardware?

6.9 Giving the IRQ and IO Address to Setserial

Once you've set the IRQ and IO address in the hardware (or arranged for it to be done by PnP) you also need to insure that the "setserial" command is run each time you start Linux. See the subsection Boot−time Configuration

6.10 High−level Configuring: stty, etc.

As a rule, your application program will do most (or all) of this. The command which does it is `stty`. See Stty

Configuring Flow Control: Hardware Flow Control is Best

See Flow Control for an explanation of it. It's usually better to use hardware flow control rather than software flow control using Xon/Xoff. To use full hardware flow control you must normally have two wires for it in the cable between the serial port and the device. If the device is on a card, then it should always be possible to use hardware flow control.

Many applications (and the getty program) give you an option regarding flow control and will set it for you. It might even set hardware flow control by default. Like the IRQ and IO address, it must be set both in the serial driver and the hardware connected to the serial port. How it's set into the hardware is hardware dependent. Often there is a certain "init string" you send to the hardware device via the serial port from your PC. For a modem, the communication program should set it in both places.

If a program you use doesn't set flow control in the serial driver, then you may do it yourself using the `stty` command. Since the driver doesn't remember the setting after you stop Linux, you could put the stty command in a file that runs at start–up or when you login (such as /etc/profile for the bash shell). Heres what you would add for hardware flow control for port ttyS2:

```
stty crtscts < /dev/ttyS2
or for stty version >= 1.17:
stty -F /dev/ttyS2 crtscts
```

`crtscts` stands for a Control setting to use the RTS and CTS pins of the serial port for hardware flow control. The upper case letters of the last sentence spell: `crtscts`.

7. Serial Port Devices /dev/ttyS2, etc.

For creating devices in the device directory see

Creating Devices In the /dev directory

7.1 Serial Port Device Names & Numbers

Devices in Linux have major and minor numbers. Each serial port may have 2 possible names in the /dev directory: ttyS and cua. Their drivers behave slightly differently. The cua device is deprecated and will not be used in the future. See the Modem–HOWTO section: "The cua Device".

Dos/Windows use the COM name while the `setserial` program uses tty00, tty01, etc. Don't confuse these with dev/tty0, dev/tty1, etc. which are used for the console (your PC monitor) but are not serial ports. The table below is for the "standard" case (but yours could be different).

```
                                                 IO
 dos            major minor       major minor  address
COM1  /dev/ttyS0  4,  64;  /dev/cua0  5,  64      3F8
COM2  /dev/ttyS1  4,  65;  /dev/cua1  5,  65      2F8
COM3  /dev/ttyS2  4,  66;  /dev/cua2  5,  66      3E8
COM4  /dev/ttyS3  4,  67;  /dev/cua3  5,  67      2E8
```

Note that all distributions should come with ttyS devices (and many distributions have the obsolete cua device). You can verify this by typing (don't feel bad if you don't find any obsolete cua devices):

```
linux% ls -l /dev/cua*
linux% ls -l /dev/ttyS*
```

7.2 Link ttySN to /dev/modem ?

On some installations, two extra devices will be created, `/dev/modem` for your modem and `/dev/mouse` for your mouse. Both of these are symbolic links to the appropriate device in `/dev` which you specified during the installation (unless you have a bus mouse, then `/dev/mouse` will point to the bus mouse device).

There has been some discussion on the merits of `/dev/mouse` and `/dev/modem`. The use of these links is discouraged. In particular, if you are planning on using your modem for dialin you may run into problems because the lock files may not work correctly if you use `/dev/modem`. However, if you change or remove this link, some applications might need reconfiguration.

7.3 Which Connector on the Back of my PC is ttyS1, etc?

Inspect the connectors

Inspecting the connectors may give some clues but is often not definitive. The serial connectors on the back side of a PC are usually DB connectors with male pins. 9−pin is the most common but some are 25−pin (especially older PCs like 486s). There may be one 9−pin (perhaps ttyS0 ??) and one 25−pin (perhaps ttyS1 ??). For two 9−pin ones the top one might be ttyS0.

If you only have one serial port connector on the back of your PC, this may be easy. If you also have an internal modem, a program like wvdial may be able to tell you what port it's on (unless it's a PnP that hasn't been PnP configured yet). A report from setserial (at boot−time or run by you from the command line) should help you identify the non−modem port.

If you have two serial connectors it may be more difficult. First check manuals (if any) for your computer. Look at the connectors for meaningful labels. You might even want to take off the PC's cover and see if there are any meaningful labels on the card where the internal ribbon cables plug in. Labels (if any) are likely to say something like "serial 1", "serial 2" or A, B. Which com port it actually is will depend on jumper or PnP settings (sometimes shown in a CMOS setup menu). But 1 or A are more likely to be ttyS0 with 2 or B ttyS1.

Send bytes to the port

Labels are not apt to be definitive so here's another method. If the serial ports have been configured correctly per setserial, then you may send some bytes out a port and try to detect which connector (if any) it's coming out of. One way to send such a signal is to copy a long text file to the port using a command like: cp my_file_name /dev/ttyS1. A voltmeter connected to the DTR pin (see Serial−HOWTO for Pinout) will display positive as soon as you give the copy command.

The transmit pin should go from several volts negative to a voltage fluctuating around zero after you start sending the bytes. If it doesn't (but the DTR went positive) then you've got the right port but it's blocked from sending. This may be due to a wrong IRQ, −clocal being set, etc. The command "`stty −F /dev/ttyS1 −a`" should show clocal (and not −clocal). If not, change it to clocal.

Another test is to jumper the transmit and receive pins (pins 2 and 3 of either the 25−pin or 9−pin connector) of a test serial port. Then send something to each port (from the PCs keyboard) and see if it gets sent back. If it does it's likely the port with the jumper on it. Then remove the jumper and verify that nothing gets sent back. Note that if "echo" is set (per stty) then a jumper creates an infinite loop. Bytes that pass thru the jumper go into the port and come right back out of the other pin back to the jumper. Then they go back in and out again and again. Whatever you send to the port repeats itself forever (until you interrupt it by removing the jumper, etc.). This may be a good way to test it as the repeating test messages halt when the jumper is removed.

As a jumper you could use a mini (or micro) jumper cable and perhaps use a scrap of paper to prevent the mini clips from accidentally touching the metal of the connector. Whatever you use as a jumper take care not to bend or excessively scratch the pins. To receive something from a port, you can go to a virtual terminal

(Alt−F2 for example) and type something like "cp /dev/ttyS2 /dev/tty". Then at another virtual terminal you may send something to ttyS2 (or whatever) by "echo test_message > /dev/ttyS2". Then go back to the receive virtual terminal and look for the test_message. See Serial Electrical Test Equipment for more info.

Connect a device to the connector

Another way to try to identify a serial port is to connect some physical serial device to it and see if it works. But a problem here is that it might not work because it's not configured right. A serial mouse might get detected if connected.

Missing connectors

If the software shows that you have more serial ports than you have connectors for (including an internal modem which counts as a serial port) then you may have a serial port that has no connector. Some motherboards come with a serial port with no cable or serial DB connector. Someone may build a PC from this and omit the connector. There may be a "serial" label on the motherboard but no ribbon cable connects to the pins next to this label. To use this port you must get a ribbon cable/connector. I've seen different wiring arrangements for such ribbon cables so beware.

7.4 Creating Devices In the /dev directory

If you don't have a device "file" that you need, you will have to create it with the `mknod` command or with the MAKEDEV shell script. Example, suppose you needed to create `ttyS0`:

```
linux# mknod −m 666 /dev/ttyS0 c 4 64
```

You can use the `MAKEDEV` script, which lives in `/dev`. See the man page for it. This simplifies the making of devices. For example, if you needed to make the devices for `ttyS0` you would type:

```
linux# cd /dev
linux# ./MAKEDEV ttyS0
```

This handles the devices creation and should set the correct permissions. For making multiport devices see Making multiport devices in the /dev directory.

---

8. Interesting Programs You Should Know About

Most info on getty has been moved to Modem−HOWTO with a little info on the use of getty with directly connected terminals now found in Text−Terminal−HOWTO.

8.1 Serial Monitoring/Diagnostics Programs

A few Linux programs (and one "file") will monitor various modem control lines and indicate if they are positive (1 or green) or negative (0 or red).

- The "file": /proc/tty/driver/serial lists those that are positive
- modemstat (Only works correctly on Linux PC consoles. Status monitored in a tiny window. Color−coded and compact. Must kill it (a process) to quit.

- statserial (Info displayed on entire screen)
- serialmon (Doesn't monitor RTS, CTS, DSR but logs other functions)

You may already have them. If not, download them from Serial Software. As of June 1998, I know of no diagnostic program in Linux for the serial port.

8.2 Changing Interrupt Priority

- `irqtune` will give serial port interrupts higher priority to improve performance.
- `hdparm` for hard−disk tuning may help some more.

8.3 What is Setserial ?

This part is in 3 HOWTOs: Modem, Serial, and Text−Terminal. There are some minor differences, depending on which HOWTO it appears in.

Introduction

Don't ever use `setserial` with Laptops (PCMCIA). `setserial` is a program which allows you to tell the device driver software the I/O address of the serial port, which interrupt (IRQ) is set in the port's hardware, what type of UART you have, etc. Since theres a good chance that the serial ports will be automatically detected and set, many people never need to use `setserial`. In any case setserial will not work without either serial support built into the kernel or loaded as a module. The module may get loaded automatically if you (or a script) tries to use setserial.

Setserial can also show how the driver is currently set. In addition, it can be made to probe the hardware and try to determine the UART type and IRQ, but this has severe limitations. See Probing. Note that it can't set the IRQ or the port address in the hardware of PnP serial ports (but the plug−and−play features of the serial driver can do this).

If you only have one or two built−in serial ports, they will usually get set up correctly without using setserial. Otherwise (or if there are problems with the serial port) you will likely need to deal with setserial. Besides the man page for `setserial`, check out info in `/usr/doc/setserial.../` or `/usr/share/doc/setserial`. It should tell you how setserial is handled in your distribution of Linux.

`Setserial` is often run automatically at boot−time by a start−up shell−script for the purpose of assigning IRQs, etc. to the driver. Setserial will only work if the serial module is loaded (or if the equivalent was compiled into your kernel). If the serial module gets unloaded later on, the changes previously made by `setserial` will be forgotten by the kernel. But recent (2000) distributions may contain scripts that save and restore this. If not, then `setserial` must be run again to reestablish them. In addition to running via a start−up script, something akin to `setserial` also runs earlier when the serial module is loaded (or the like). Thus when you watch the start−up messages on the screen it may look like it ran twice, and in fact it has.

Setserial can set the time that the port will keep operating after it's closed (in order to output any characters still in its buffer in main RAM). This is needed at slow baud rates of 1200 or lower. It's also needed at higher speeds if there are a lot of "flow control" waits. See "closing_wait" in the man pg.

Setserial does not set either IRQ's nor I/O addresses in the serial port hardware itself. That is done either by jumpers or by plug−and−play. You must tell setserial the identical values that have been set in the hardware. Do not just invent some values that you think would be nice to use and then tell them to setserial. However, if you know the I/O address but don't know the IRQ you may command setserial to attempt to determine the IRQ.

You can see a list of possible commands by just typing `setserial` with no arguments. This fails to show you the one−letter options such as −v for verbose which you should normally use when troubleshooting. Note that setserial calls an IO address a "port". If you type:

```
setserial −g /dev/ttyS*
```

you'll see some info about how that device driver is configured for your ports. Note that where it says **"`UART: unknown`"** it probably means that no uart exists. In other words you probably have no such serial port and the other info shown about the port is meaningless and should be ignored. If you really do have such a serial port, setserial doesn't recognize it and that needs to be fixed.

If you add −a to the option −g you will see more info although few people need to deal with (or understand) this additional info since the default settings you see usually work fine. In normal cases the hardware is set up the same way as "setserial" reports, but if you are having problems there is a good chance that "setserial" has it wrong. In fact, you can run "setserial" and assign a purely fictitious I/O port address, any IRQ, and whatever uart type you would like to have. Then the next time you type "setserial ..." it will display these bogus values without complaint. They will also be officially registered with the kernel as seen by the "scanport" command. Of course the serial port driver will not work correctly (if at all) if you attempt to use such a port. Thus when giving parameters to "setserial" anything goes. Well almost. If you assign one port a base address that is already assigned (such as 3e8) it will not accept it. But if you use 3e9 it will accept it. Unfortunately 3e9 is already assigned since it is within the range starting at base address 3e8. Thus the moral of the story is to make sure of your data before assigning resources with setserial.

While assignments made by setserial are lost when the PC is powered off, a configuration file may restore them (or a previous configuration) when the PC is started up again. In newer versions, what you change by setserial gets automatically saved to a configuration file. In older versions, the configuration file only changes if you edit it manually so the configuration remains the same from boot to boot. See Configuration Scripts/Files

Probing

With appropriate options, `setserial` can probe (at a given I/O address) for a serial port but you must guess the I/O address. If you ask it to probe for /dev/ttyS2 for example, it will only probe at the address it thinks ttyS2 is at (2F8). If you tell setserial that ttyS2 is at a different address, then it will probe at that address, etc. See Probing

The purpose of this is to see if there is a uart there, and if so, what its IRQ is. Use "setserial" mainly as a last resort as there are faster ways to attempt it such as wvdialconf to detect modems, looking at very early boot−time messages, or using `pnpdump −−dumpregs`. To try to detect the physical hardware use the −v (verbose) and `autoconfig` command to `setserial`. If the resulting message shows a uart type such as 16550A, then you're OK. If instead it shows "`unknown`" for the uart type, then there is supposedly no serial port at all at that I/O address. Some cheap serial ports don't identify themselves correctly so if you see "`unknown`" you still might have a serial port there.

Besides auto−probing for a uart type, setserial can auto−probe for IRQ's but this doesn't always work right either. In one case it first gave the wrong irq but when the command was repeated it found the correct irq. In versions of setserial >= 2.15, the results of your last probe test may be saved and put into the configuration file `/etc/serial.conf` which will be used next time you start Linux. At boot−time when the serial module loads (or the like), a probe for UARTs is made automatically and reported on the screen. But the IRQs shown may be wrong. The second report of the same is the result of a script which usually does no probing and thus provides no reliable information as to how the hardware is actually set. It only shows configuration data someone wrote into the script or data that got saved in /etc/serial.conf.

It may be that two serial ports both have the same IO address set in the hardware. Of course this is not permitted but it sometimes happens anyway. Probing detects one serial port when actually there are two. However if they have different IRQs, then the probe for IRQs may show IRQ = 0. For me it only did this if I first used `setserial` to give the IRQ a ficticious value.

Boot−time Configuration

When the kernel loads the serial module (or if the "module equivalent" is built into the kernel) then only `ttyS{0−3}` are auto−detected and the driver is set to use only IRQs 4 and 3 (regardless of what IRQs are actually set in the hardware). You see this as a boot−time message just like as if `setserial` had been run.

To correct possible errors in IRQs (or for other reasons) there may be a file somewhere that runs `setserial` again. Unfortunately, if this file has some IRQs wrong, the kernel will still have incorrect info about the IRQs. This file should run early at boot−time before any process uses the serial port. In fact, your distribution may have set things up so that the setserial program runs automatically from a start−up script at boot−time. More info about how to handle this situation for your particular distribution might be found in file named "setserial..." or the like located in directory /usr/doc/ or /usr/share/doc/.

Before modifying a configuration file, you can test out a "proposed" `setserial` command by just typing it on the command line. In some cases the results of this use of `setserial` will automatically get saved in /etc/serial.conf when you shutdown. So if it worked OK (and solved your problem) then there's no need to modify any configuration file. See New configuration method using /etc/serial.conf.

Configuration Scripts/Files

Your objective is to modify (or create) a script file in the /etc tree that runs setserial at boot−time. Most distributions provide such a file (but it may not initially reside in the /etc tree). In addition, setserial 2.15 and higher often have an /etc/serial.conf file that is used by the above script so that you don't need to directly edit the script that runs setserial. In addition just using setserial on the command line (2.15+) may ultimately alter this configuration file.

So prior to version 2.15 all you do is edit a script. After 2.15 you may need to either do one of three things: 1. edit a script. 2. edit `/etc/serial.conf` or 3. run "setserial" on the command line which may result in `/etc/serial.conf` automatically being edited. Which one of these you need to do depends on both your particular distribution, and how you have set it up.

Edit a script (required prior to version 2.15)

Prior to setserial 2.15 (1999) there was no /etc/serial.conf file to configure setserial. Thus you need to find the file that runs "setserial" at boot time and edit it. If it doesn't exist, you need to create one (or place the commands in a file that runs early at boot−time). If such a file is currently being used it's likely somewhere in the /etc directory−tree. But Redhat <6.0 has supplied it in /usr/doc/setserial/ but you need to move it to the /etc tree before using it. You might use "locate" to try to find such a file. For example, you could type: locate "*serial*".

The script `/etc/rc.d/rc.serial` was commonly used in the past. The Debian distribution used `/etc/rc.boot/0setserial`. Another file once used was `/etc/rc.d/rc.local` but it's not a good idea to use this since it may not be run early enough. It's been reported that other processes may try to open the serial port before rc.local runs resulting in serial communication failure. Today it's most likely in /etc/init.d/ but it isn't normally intended to be edited.

If such a file is supplied, it should contain a number of commented−out examples. By uncommenting some of these and/or modifying them, you should be able to set things up correctly. Make sure that you are using a valid path for `setserial`, and a valid device name. You could do a test by executing this file manually (just type its name as the super−user) to see if it works right. Testing like this is a lot faster than doing repeated reboots to get it right.

For versions >= 2.15 (provided your distribution implemented the change, Redhat didn't) it may be more tricky to do since the file that runs setserial on startup, /etc/init.d/setserial or the like was not intended to be edited by the user. See New configuration method using /etc/serial.conf.

If you want setserial to automatically determine the uart and the IRQ for ttyS3 you would add something like:

```
/sbin/setserial  /dev/ttyS3 auto_irq skip_test autoconfig
```

Do this for every serial port you want to auto configure. Be sure to give a device name that really does exist on your machine. In some cases this will not work right due to the hardware. If you know what the uart and irq actually are, you may want to assign them explicitly with "setserial". For example:

```
/sbin/setserial /dev/ttyS3 irq 5 uart 16550A  skip_test
```

New configuration method using /etc/serial.conf

Prior to setserial version 2.15, the way to configure setserial was to manually edit the shell−script that ran setserial at boot−time. See Edit a script (after version 2.15: perhaps not). Starting with version 2.15 (1999) of `setserial` this shell−script is not edited but instead gets its data from a configuration file: `/etc/serial.conf`. Furthermore you may not even need to edit serial.conf because using the "setserial" command on the command line may automatically cause serial.conf to be edited appropriately.

This was intended to make it so that you don't need to edit any file in order to set up (or change) setserial so it will do the right thing each time that Linux is booted. But there are serious pitfalls because it's not really "setserial" that edits serial.conf. Confusion is compounded because different distributions handle this differently. In addition, you may modify it so it works differently.

What often happens is this: When you shut down your PC the script that runs "setserial" at boot−time is run again, but this time it only does what the part for the "stop" case says to do: It uses "setserial" to find out what the current state of "setserial" is and puts that info into the `serial.conf` file. Thus when you run "setserial" to change the serial.conf file, it doesn't get changed immediately but only when and if you shut down normally.

Now you can perhaps guess what problems might occur. Suppose you don't shut down normally (someone turns the power off, etc.) and the changes don't get saved. Suppose you experiment with "setserial" and forget to run it a final time to restore the original state (or make a mistake in restoring the original state). Then your "experimental" settings are saved.

If you manually edit serial.conf, then your editing is destroyed when you shut down because it gets changed back to the state of setserial at shutdown. There is a way to disable the changing of serial.conf at shutdown and that is to remove "###AUTOSAVE###" or the like from first line of serial.conf. In at least one distribution, the removal of "###AUTOSAVE###" from the first line is automatically done after the first time you shutdown just after installation. The serial.conf file will hopefully contain some comments to help you out.

The file most commonly used to run setserial at boot−time (in conformance with the configuration file) is now /etc/init.d/setserial (Debian) or /etc/init.d/serial (Redhat), or etc., but it should not normally be edited. For 2.15 Redhat 6.0 just had a file /usr/doc/setserial−2.15/rc.serial which you have to move to /etc/init.d/ if you want setserial to run at boot−time.

To disable a port, use `setserial` to set it to "uart none". The format of /etc/serial.conf appears to be just like that of the parameters placed after "setserial" on the command line with one line for each port. If you don't use autosave, you may edit /etc/serial.conf manually.

BUG: As of July 1999 there is a bug/problem since with ###AUTOSAVE### only the setserial parameters displayed by "setserial −Gg /dev/ttyS*" get saved but the other parameters don't get saved. Use the −a flag to "setserial" to see all parameters. This will only affect a small minority of users since the defaults for the parameters not saved are usually OK for most situations. It's been reported as a bug and may be fixed by now.

In order to force the current settings set by setserial to be saved to the configuration file (serial.conf) without shutting down, do what normally happens when you shutdown: Run the shell−script `/etc/init.d/{set}serial stop`. The "stop" command will save the current configuration but the serial ports still keep working OK.

In some cases you may wind up with both the old and new configuration methods installed but hopefully only one of them runs at boot−time. Debian labeled obsolete files with "...pre−2.15".

IRQs

By default, both ttyS0 and ttyS2 will share IRQ 4, while ttyS1 and ttyS3 share IRQ 3. But actually sharing serial interrupts (using them in running programs) is not permitted unless you: 1. have kernel 2.2 or better, and 2. you've complied in support for this, and 3. your serial hardware supports it. See

Interrupt sharing and Kernels 2.2+

If you only have two serial ports, ttyS0 and ttyS1, you're still OK since IRQ sharing conflicts don't exist for non−existent devices.

If you add an internal modem and retain ttyS0 and ttyS1, then you should attempt to find an unused IRQ and set it both on your serial port (or modem card) and then use setserial to assign it to your device driver. If IRQ 5 is not being used for a sound card, this may be one you can use for a modem. To set the IRQ in hardware you may need to use isapnp, a PnP BIOS, or patch Linux to make it PnP. To help you determine which spare IRQ's you might have, type "man setserial" and search for say: "IRQ 11".

8.4 Stty

Introduction

`stty` does much of the configuration of the serial port but since application programs (and the getty program) often handle it, you may not need to use it much. It's handy if your having problems or want to see how the port is set up. Try typing ``stty −a'' at your terminal/console to see how it's now set. Also try typing it without the −a (all) for a short listing which shows how it's set different than normal. Don't try to learn all the setting unless you want to become a serial guru. Most of the defaults should work OK and some of the settings are needed only for certain obsolete dumb terminals made in the 1970's.

`stty` is documented in the man pages with a more detailed account in the info pages. Type `"man stty"` or `"info stty"`.

Whereas `setserial` only deals with actual serial ports, stty is used both for serial ports and for virtual terminals such as the standard Linux text interface at a PC monitor. For the PC monitor, many of the stty settings are meaningless. Changing the baud rate, etc. doesn't appear to actually do anything.

Here are some of the items stty configures: speed (bits/sec), parity, bits/byte, # of stop bits, strip 8th bit?, modem control signals, flow control, break signal, end−of−line markers, change case, padding, beep if buffer overrun?, echo what you type to the screen, allow background tasks to write to terminal?, define special (control) characters (such as what key to press for interrupt). See the `stty` man or info page for more details. Also see the man page: `termios` which covers the same options set by stty but (as of mid 1999) covers features which the stty man page fails to mention.

With some implementations of getty (getty_ps package), the commands that one would normally give to stty are typed into a getty configuration file: /etc/gettydefs. Even without this configuration file, the getty command line may be sufficient to set things up so that you don't need stty.'")

One may write C programs which change the stty configuration, etc. Looking at some of the documentation for this may help one better understand the use of the stty command (and its many possible arguments). Serial−Programming−HOWTO is useful. The manual page: termios contains a description of the C−language structure (of type termios) which stores the stty configuration in computer memory. Many of the flag names in this C−structure are almost the same (and do the same thing) as the arguments to the stty command.

Using stty at a "foreign" terminal

Using `stty` to configure the terminal that you are currently using is easy. Doing it for a different (foreign) terminal or serial port may be tricky. For example, let's say you are at the PC monitor (tty1) and want to use `stty` to deal with the serial port ttyS2. Prior to about 2000 you needed to use the redirection operator "<". After 2000 (provided your version of setserial is >= 1.17 and stty >= 2.0) there is a better method using the −F option. This will work when the old redirection method fails. Even with the latest versions be warned that if there is a terminal on ttyS2 and a shell is running on that terminal, then what you see will likely be deceptive and trying to set it will not work. See Two interfaces at a terminal to understand it.

The new method is ``stty −F /dev/ttyS2 ...'' (or −−file instead of F). If ... is −a it displays all the stty settings. The old redirection method (which still works in later versions) is to type ``stty ... </dev/ttyS2''. If the new method works but the old one hangs, it implies that the port is hung due to lack of a modem control line from being asserted. Thus the old method is still useful for troubleshooting. See the following subsection.

Old redirection method

Here's a problem with the old redirection operator (which doesn't happen if you use the newer −F option instead). Sometimes when trying to use stty, the command hangs and nothing happens (you don't get a prompt for a next command even after hitting <return>). This is likely due to the port being stuck because it's waiting for one of the modem control lines to be asserted. For example, unless you've set "clocal" to ignore modem control lines, then if no CD signal is asserted the port will not open and stty will not work for it (unless you use the newer −F option). A similar situation seems to exist for hardware flow control. If the cable for the port doesn't even have a conductor for the pin that needs to be asserted then there is no easy way to stop the hang.

One way to try to get out of the above hang is to use the newer −F option and set "clocal" and/or "crtscts". If you don't have the −F option then you may try to run program on the port that will force it to operate even if the control lines say not to. Then hopefully this program might set the port so it doesn't need the control signal in the future in order to open: clocal or −crtscts. To use "minicom" to do this you have to reconfigure minicom for another ttyS, etc, and then exit it and restart it. Since you then have to reconfigure minicom again, it may be simpler to just reboot the PC.

The old redirection method makes ttyS2 the standard input to stty. This gives the stty program a link to the "file" ttyS2 so that it may "read" it. But instead of reading the bytes sent to ttyS2 as one might expect, it uses the link to find the configuration settings of the port so that it may read or change them. Some people tried to use ``stty ... > /dev/ttyS2" to set the terminal. This will not do it. Instead, it takes the message normal displayed by the stty command for the terminal you are on (say tty1) and sends this message to ttyS2. But it doesn't change any settings for ttyS2.

Two interfaces at a terminal

When using a shell (such as bash) with command−line−editing enabled there are two different terminal interfaces (what you see when you type stty −a). When you type in modern shells at the command line you have a temporary "raw" interface (or raw mode) where each character is read by the command−line−editor as you type it. Once you hit the <return> key, the command−line−editor is exited and the terminal interface is changed to the nominal "cooked" interface (cooked mode) for the terminal. This cooked mode lasts until the next prompt is sent to the terminal. Note that one never gets to type anything to this cooked mode but what was typed in raw mode becomes cooked mode as soon as one hits the <return> key.

When a prompt is sent to the terminal the terminal goes from "cooked" to "raw" mode (just like it does when you start an editor since you are starting the command−line editor). The settings for the "raw" mode are based only on the basic settings taken from the "cooked" mode. Raw mode keeps these setting but changes several other settings in order to change the mode to "raw". It is not at all based on the settings used in the previous "raw" mode. Thus if one uses stty to change settings for the raw mode, such settings will be lost as soon as one hits the <return> key at the terminal that has supposedly been "set".

Now when one types stty to look at the terminal interface, one may either get a view of the cooked mode or the raw mode. You need to figure out which one you're looking at. It you use stty from another terminal to deal with a terminal that is displaying a command line, then you will see the raw mode settings. Any changes made will only be made to the raw mode and will be lost when someone presses <return> at the terminal you tried to "set". But if you type a stty command at your terminal (without using < for redirection) and then hit <return> it's a different story. The <return> puts the terminal in cooked mode. Your changes are saved and will still be there when the terminal goes back into raw mode (unless of course it's a setting not allowed in raw mode).

This situation can create problems. For example, suppose you corrupt your terminal interface and to restore it you go to another terminal and "stty −F dev/ttyS1 sane" (or the like) to restore it. It will not work! Of course you can try to type "stty sane ..." at the terminal that is corrupted but you can't see what you typed. All the above not only applies to dumb terminals but to virtual terminals used on a PC Monitor as well as to the terminal windows in X. In other words, it applies to almost everyone who uses Linux.

Luckily, when you start up Linux, any file that runs stty at boot−time will likely deal with a terminal (or serial port with no terminal) that has no shell running on it so there's no problem for this special case.

Where to put the stty command ?

Should you need to have `stty` set up the serial interface each time the computer starts up then you need to put the `stty` command in a file that will be executed each time the computer is started up (Linux boots). It should be run before the serial port is used (including running getty on the port). There are many possible places to put it. If it gets put in more than one place and you only know about (or remember) one of those places, then a conflict is likely. So make sure to document what you do.

One place to put it would be in the same file that runs setserial when the system is booted. The location is distribution and version dependent. It would seem best to put it after the setserial command so that the low level stuff is done first. If you have directories in the /etc tree where every file in them is executed at boot−time (System V Init) then you could create a file named "stty" for this purpose.

8.5 What is isapnp ?

`isapnp` is a program to configure Plug−and−Play (PnP) devices on the ISA bus including internal modems. It comes in a package called "isapnptools" and includes another program, "pnpdump" which finds all your ISA PnP devices and shows you options for configuring them in a format which may be added to the PnP configuration file: /etc/isapnp.conf. The isapnp command may be put into a startup file so that it runs each time you start the computer and thus will configure ISA PnP devices. It is able to do this even if your BIOS doesn't support PnP. See Plug−and−Play−HOWTO.

---

9. Speed (Flow Rate)

By "speed" we really mean the "data flow rate" but almost everybody incorrectly calls it speed. The speed is measured in bits/sec (or baud). Speed is set using the "stty" command or by a program which uses the serial port. See Stty

9.1 Can't Set a High Enough Speed

You need to find out the highest speed supported by your hardware. As of late 1998 (and late 2000) most serial ports only supported speeds up to 115.2k bps. Some 56k internal modems support 230.4k bps (but it may be hard to find out which ones do). Recent Linux kernels support high speeds (over 115.2k) but you might have some problems using it because of one or both of the following reasons:

1. Older versions of an application program (or stty) may not accept the high speed.
2. Setserial may show the wrong speed unless you use the baud_base option. In any case the high speed will work OK.

How speed is set in hardware: the divisor and baud_base

Here's a list of commonly used divisors and their corresponding speeds (assuming a maximum speed of 115,200): 1 (115.2k), 2 (57.6k), 3 (38.4k), 6 (19.2k), 12 (9.6k), 24 (4.8k), 48 (2.4k), 96 (1.2k), etc. The serial driver sets the speed in the hardware by sending the hardware only a "divisor" (a positive integer). This "divisor" divides the maximum speed of the hardware resulting in a slower speed (except a divisor of 1 obviously tells the hardware to run at maximum speed).

Normally, if you specify a speed of 115.2k (in your communication program or by stty) then the serial driver sets the port hardware to divisor 1 which obviously sets the highest speed. If you happen to have hardware with a maximum speed of say 230.4k, then specifying 115.2k will result in divisor 1 and will actually give you 230.4k. This is double the speed that you set. In fact, for any speed you set, the actual speed will be double. If you had hardware that could run at 460.8k then the actual speed would be quadruple what you set.

Work−arounds for setting speed

To correct this accounting (but not always fix the problem) you may use "setserial" to change the baud_base to the actual maximal speed of your port such as 230.4k. Then if you set the speed (by your application or by stty) to 230.4k, a divisor of 1 will be used and you'll get the same speed as you set. PROBLEM: stty and many communication programs (as of mid 1999) still have 115.2k as their maximum speed setting and will not let you set 230.4k, etc. So in these cases one solution is not to change anything with `setserial` but mentally keep in mind that the actual speed is always double what you set.

There's another work−around which is not much better. To use it you set the baud_base (with setserial) to the maximal speed of your hardware. This corrects the accounting so that if you set say 115.2k you actually get 115.2k. Now you still have to figure out how to set the highest speed if your communication program (or the like) will not let you do it. Fortunately, setserial has a way to do this: use the "spd_cust" parameter with "divisor 1". Then when you set the speed to 38400 in a communication program, the divisor will be set to 1 in the port and it will operate at maximum speed. For example:
setserial /dev/ttyS2 spd_cust baud_base 230400 divisor 1
Don't try using "divisor" for any other purpose other than the special use illustrated above (with spd_cust).

If there are two or more high speeds that you want to use that your communication program can't set, then it's not quite as easy as above. But the same principles apply. You could just keep the default baud_base and understand that when you set a speed you are really only setting a divisor. So your actual speed will always be your maximum speed divided by whatever divisor is set by the serial driver. See How speed is set in hardware: the divisor and baud_base

Crystal frequency is not baud_base

Note that the baud_base setting is usually much lower than the frequency of the crystal oscillator in the hardware since the crystal frequency is often divided by 16 in the hardware to get the actual top speed. The reason the crystal frequency needs to be higher is so that this high crystal speed can be used to take a number of samples of each bit to determine if it's a 1 or a 0.

9.2 Higher Serial Throughput

If you are seeing slow throughput and serial port overruns on a system with (E)IDE disk drives, you can get `hdparm`. This is a utility that can modify (E)IDE parameters, including unmasking other IRQs during a disk IRQ. This will improve responsiveness and will help eliminate overruns. Be sure to read the man page very carefully, since some drive/controller combinations don't like this and may corrupt the filesystem.

Also have a look at a utility called `irqtune` that will change the IRQ priority of a device, for example the serial port that your modem is on. This may improve the serial throughput on your system. The `irqtune` FAQ is at http://www.best.com/~cae/irqtune

---

10. Locking Out Others

10.1 Introduction

When you are using a serial port, you may want to prevent others from using it at the same time. However there may be cases where you do want others to use it, such as sending you an important message if you are using a text−terminal.

There are various ways of preventing others (or other processes) from using your serial port when you are using it (locking). This should all happen automatically but it's important to know about this if it gives you trouble. If a program is abnormally exited or the PC is abruptly turned off (by pulling the plug, etc.) your serial port might wind up locked. Even if the lock remains, it's usually automatically removed when you want to use the serial port again. But in rare cases it isn't. That's when you need to understand what happened.

One way to implement locking is to design the kernel to handle it but Linux thus far has shunned this solution (with an exception involving the cua device which is now obsolete). Two solutions used by Linux is to:

1. create lock−files
2. modify the permissions and/or owners of devices such as /dev/ttyS2

10.2 Lock−Files

A lock−file is simply a file created to mean that a particular device is in use. They are kept in `/var/lock`. Formerly they were in `/usr/spool/uucp`. Linux lock−files are sometimes named `LCK..`*name*, where *name* is either a device name, or a UUCP site name. Most processes (an exception is getty) create these locks so that they can have exclusive access to devices. For instance if you dial out on your modem, a lock−file (or even more that one lockfile) will appear telling other processes that someone else is using the modem. Lock files contain the PID of the process that has locked the device. Note that if a process insists on using a device that is locked, it may ignore the lockfile and use the device anyway. This is useful in sending a message to a text−terminal, etc.

When a program wants to use a serial port but finds it locked with a lock−file it should check to see if the lock−file's PID is still in use. If it's not it means that the lock is stale and it's OK to go ahead and use the port anyway (after removing the stale lock−file). Unfortunately, there may be some programs that don't do this and give up by telling you that a device is already in use when it really isn't.

Problems can arise with lockfiles if the same device has two different names, resulting in lockfiles with different names that actually are the same device. Formerly each physical serial port was known by two different device names: ttyS0 and cua0. The lock checking software is aware of ttyS vs. cua but it's simpler now since cua has been eliminated. Older versions may still use cua. Using alternate names (such as /dev/modem for /dev/ttyS2) is asking for trouble.

For dumb terminals, lockfiles are not used since this would not permit someone else to send a message to your terminal using the write or talk program.

10.3 Change Owners, Groups, and/or Permissions of Device Files

In order to use a device, you (or the program you run if you have "set user id") needs to have permission to read and write the device "file" in the /dev directory. So a logical way to prevent others from using a device is to make yourself the temporary owner of the device and set permissions so that no one else can use it. A program may do this for you. A similar method can be used with the group of the device file.

While lock files prevent other process from using the device, changing device file owners/permissions restricts other users (or the group) from using it. One case is where the group is permitted to write to the port, but not to read from it. Writing to the port might just mean a message sent to a text−terminal while reading means destructive reading. The original process that needs to read the data may find data missing if another process has already read that data. Thus a read can do more harm that a write since a read causes loss of data while a write only adds extra data. That's a reason to allow writes but not reads. This is exactly the opposite of the case for ordinary files where you allow others to read the file but not write (modify) it. Use of a port normally requires both read and write permissions.

A program that changes the device file attributes should undo these changes when it exits. But if the exit is abnormal, then a device file may be left in such a condition that it gives the error "permission denied" when one attempts to use it again.

---

11. Communications Programs And Utilities

11.1 List of Software

Here is a list of some communication software you can choose from, available via FTP, if they didn't come with your distribution.

- `ecu` – a communications program
- C−Kermit – portable, scriptable, serial and TCP/IP communications including file transfer, character−set translation, and zmodem support
- `gkermit` Tiny GPLed kermit run only from the command line. Can't connect to another computer
- `minicom` – telix−like communications program
- `seyon` – X based communication program
- `xc` – xcomm communication package
- `term` and `SLiRP` offer TCP/IP functionality using a shell account.
- `screen` is another multi−session program. This one behaves like the virtual consoles.
- `callback` is where you dial out to a remote modem and then that modem hangs up and calls you back (to save on phone bills).
- `mgetty+fax` handles FAX stuff, and provides an alternate `ps_getty`.
- `ZyXEL` is a control program for ZyXEL U−1496 modems. It handles dialin, dialout, dial back security, FAXing, and voice mailbox functions.
- SLIP and PPP software can be found at `ftp://metalab.unc.edu/pub/Linux/system/network/serial`.

11.2 kermit and zmodem

For use of kermit with modems see the Modem−HOWTO. One can run zmodem within the kermit program. To do this (for ttyS3), add the following to your `.kermrc` file:

```
define rz !rz < /dev/ttyS3 > /dev/ttyS3
define sz !sz \%0 > /dev/ttyS3 < /dev/ttyS3
```

Be sure to put in the correct port your modem is on. Then, to use it, just type `rz` or `sz <filename>` at the `kermit` prompt.

---

12. Serial Tips And Miscellany

12.1 Serial Module

Often the serial driver is provided as a module. Parameters may be supplied to certain modules in /etc/modules.conf. Since kernel 2.2 you don't edit this file but use the program update−modules to change it. The info that is used to update modules.conf is put in /etc/modutils/. The Debian/GNU Linux has a file here named /etc/modutils/setserial which runs the serial script in /etc/init.d/ every time the serial module is loaded or unloaded. When the serial module is unloaded this script will save the state of the module in /var/run/setserial.conf. Then if the module loads again this saved state is restored. When the serial module first loads at boot−time, there's nothing in /var/run/setserial.conf so the state is obtained from /etc/serial.conf. So there are two files that save the state. Other distributions may do something similar.

One may modify the serial driver by editing the source code. Much of the serial driver is found in the file serial.c. For info regarding writing of programs for the serial port see Serial−Programming−HOWTO (revised in 1999 by Vern Hoxie but not at LDP. Get it from scicom.alphacdc.com/pub/linux)

12.2 Serial Console (console on the serial port)

See the kernel documentation in: Documentation/serial−console.txt. See also "Serial Console" in Text−Terminal−HOWTO.

12.3 Line Drivers

For a text terminal, the EIA−232 speeds are fast enough but the usable cable length is often too short. Balanced technology could fix this. The common method of obtaining balanced communication with a text terminal is to install 2@ line drivers in the serial line to convert unbalanced to balanced (and conversely). They are a specialty item and are expensive if purchased new.

12.4 Known Defective Hardware

Avoiding IO Address Conflicts with Certain Video Boards

The IO address of the IBM 8514 video board (and others) is allegedly 0x?2e8 where ? is 2, 4, 8, or 9. This may conflict (but shouldn't if the serial port is well designed) with the IO address of `ttyS3` at 0x02e8 if the serial port ignores the leading 0 hex digit when it decodes the address (many do). That is bad news if you try to use `ttyS3` at this IO address. Another story is that Linux will not detect your internal modem on `ttyS3` but that you can use `setserial` to put `ttyS3` at this address and the modem will work fine.

Problem with AMD Elan SC400 CPU (PC−on−a−chip)

This has a race condition between an interrupt and a status register of the UART. An interrupt is issued when the UART transmitter finishes the transmission of a byte and the UART transmit buffer becomes empty (waiting for the next byte). But a status register of the UART doesn't get updated fast enough to reflect this. As a result, the interrupt service routine rapidly checks and determines (erroneously) that nothing has happened. Thus no byte is sent to the port to be transmitted and the UART transmitter waits in vain for a byte that never arrives. If the interrupt service routine had waited just a bit longer before checking the status register, then it would have been updated to reflect the true state and all would be OK.

There is a proposal to fix this by patching the serial driver. But Should linux be patched to accommodate defective hardware, especially if this patch may impair performance of good hardware?

---

13. Troubleshooting

See Modem−HOWTO for troubleshooting related to modems or getty for modems. For a Text−Terminal much of the info here will be of value as well as the troubleshooting info in Text−Terminal−HOWTO.

13.1 Serial Electrical Test Equipment

Breakout Gadgets, etc.

While a multimeter (used as a voltmeter) may be all that you need for just a few serial ports, simple special test equipment has been made for testing serial port lines. Some are called "breakout ... " where breakout means to break out conductors from a cable. These gadgets have a couple of connectors which connect to serial port connectors (either at the ends of serial cables or at the back of a PC). Some have test points for connecting a voltmeter. Others have LED lamps which light when certain modem control lines are asserted (turned on). The color of the light may indicate the polarity of the signal (positive or negative voltage). Still others have jumpers so that you can connect any wire to any wire. Some have switches.

Radio Shack sells (in 1998) a "RS−232 Troubleshooter" or "RS−232 Line Tester" which checks TD, RD, CD, RTS, CTS, DTR, and DSR. A green light means on (+12 v) while red means off (−12 v). They also sell a "RS−232 Serial Jumper Box" which permits connecting the pins anyway you choose.

Measuring Voltages

Any voltmeter or multimeter, even the cheapest that sells for about $10, should work fine. Trying to use other methods for checking voltage is tricky. Don't use a LED unless it has a series resistor to reduce the voltage across the LED. A 470 ohm resistor is used for a 20 ma LED (but not all LED's are 20 ma). The LED will only light for a certain polarity so you may test for + or − voltages. Does anyone make such a gadget for automotive circuit testing?? Logic probes may be damaged if you try to use them since the TTL voltages for which they are designed are only 5 volts. Trying to use a 12 V incandescent light bulb is not a good idea. It won't show polarity and due to limited output current of the UART it probably will not even light up.

To measure voltage on a female connector you may plug in a bent paper clip into the desired opening. The paper clip's diameter should be no larger than the pins so that it doesn't damage the contact. Clip an alligator clip (or the like) to the paper clip to connect up. Take care not to touch two pins at the same time with any metal object.

Taste Voltage

As a last resort, if you have no test equipment and are willing to risk getting shocked (or even electrocuted) you can always taste the voltage. Before touching one of the test leads with your tongue, test them to make sure that there is no high voltage on them. Touch both leads (at the same time) to one hand to see if they shock you. Then if no shock, wet the skin contact points by licking and repeat. If this test gives you a shock, you certainly don't want to use your tongue.

For the test for 12 V, Lick a finger and hold one test lead in it. Put the other test lead on your tongue. If the lead on your tongue is positive, there will be a noticeable taste. You might try this with flashlight batteries first so you will know what taste to expect.

13.2 Serial Monitoring/Diagnostics

A few Linux programs will monitor the modem control lines and indicate if they are positive (1) or negative (0). See section Serial Monitoring/Diagnostics

13.3 (The following subsections are in both the Serial and Modem HOWTOs)

13.4 My Serial Port is Physically There but Can't be Found

If a physical device (such as a modem) doesn't work at all it may mean that the device is not at the I/O address that setserial thinks it's at. It could also mean (for a PnP card) that is doesn't yet have an address. Thus it can't be found.

Check the BIOS menus and BIOS messages. For the PCI bus use lspci or scanpci. If it's an ISA bus PnP serial port, try "pnpdump --dumpregs" and/or see Plug-and-Play-HOWTO. Using "scanport" will scan all ISA bus ports and may discover an unknown port that could be a serial port (but it doesn't probe the port). It could hang your PC. You may try probing with setserial. See Probing. If nothing seems to get thru the port it may be accessible but have a bad interrupt. See Extremely Slow: Text appears on the screen slowly after long delays. Use `setserial -g` to see what the serial driver thinks and check for IRQ and I0 address conflicts. Even if you see no conflicts the driver may have incorrect information (view it by "setserial" and conflicts may still exist.

If two ports have the same IO address then probing it will erroneously indicate only one port. Plug-and-play detection will find both ports so this should only be a problem if at least one port is not plug-and-play. All sorts of errors may be reported/observed for devices illegally "sharing" a port but the fact that there are two devices on the same a port doesn't seem to get detected (except hopefully by you). In the above case, if the IRQs are different then probing for IRQs with setserial might "detect" this situation by failing to detect any IRQ. See Probing.

13.5 Extremely Slow: Text appears on the screen slowly after long delays

It's likely mis−set/conflicting interrupts. Here are some of the symptoms which will happen the first time you try to use a modem, terminal, or serial printer. In some cases you type something but nothing appears on the screen until many seconds later. Only the last character typed may show up. It may be just an invisible <return> character so all you notice is that the cursor jumps down one line. In other cases where a lot of data should appear on the screen, only a batch of about 16 characters appear. Then there is a long wait of many seconds for the next batch of characters. You might also get "input overrun" error messages (or find them in logs).

For more details on the symptoms and why this happens see

Interrupt Problem Details and/or Interrupt Conflicts and/or Mis−set Interrupts. If it involves Plug−and−Play devices, see also Plug−and−Play−HOWTO.

As a quick check to see if it really is an interrupt problem, set the IRQ to 0 with "setserial". This will tell the driver to use polling instead of interrupts. If this seems to fix the "slow" problem then you had an interrupt problem. You should still try to solve the problem since polling uses excessive computer resources.

Checking to find the interrupt conflict may not be easy since Linux supposedly doesn't permit any interrupt conflicts and will send you a /dev/ttyS?: Device or resource busy error message if it thinks you are attempting to create a conflict. But a real conflict can be created if "setserial" has told the kernel incorrect info. The kernel has been lied to and thus doesn't think there is any conflict. Thus using "setserial" will not reveal the conflict (nor will looking at /proc/interrupts which bases its info on "setserial"). You still need to know what "setserial" thinks so that you can pinpoint where it's wrong and change it when you determine what's really set in the hardware.

What you need to do is to check how the hardware is set by checking jumpers or using PnP software to check how the hardware is actually set. For PnP run either "pnpdump −−dumpregs" (if ISA bus) or run "lspci" (if PCI bus). Compare this to how Linux (e.g. "setserial") thinks the hardware is set.

13.6 Somewhat Slow: I expected it to be a few times faster

One reason may be that whatever is on the serial port (such as a modem, terminal, printer) doesn't work as fast as you thought it did.

Another possible reason is that you have an obsolete serial port: UART 8250, 16450 or early 16550 (or the serial driver thinks you do). See

What Are UARTS? Use "setserial −g /dev/ttyS*". If it shows anything less than a 16550A, this may be your problem. If you think that "setserial" has it wrong check it out. See What is Setserial for more info. If you really do have an obsolete serial port, lying about it to setserial will only make things worse.

13.7 The Startup Screen Show Wrong IRQs for the Serial Ports.

Linux does not do any IRQ detection on startup. When the serial module loads it only does serial device detection. Thus, disregard what it says about the IRQ, because it's just assuming the standard IRQs. This is done, because IRQ detection is unreliable, and can be fooled. But if and when setserial runs from a start−up script, it changes the IRQ's and displays the new (and hopefully correct) state on on the startup screen. If the wrong IRQ is not corrected by a later display on the screen, then you've got a problem.

So, even though I have my `ttyS2` set at IRQ 5, I still see

```
ttyS02 at 0x03e8 (irq = 4) is a 16550A
```

at first when Linux boots. (Older kernels may show "ttyS02" as "tty02" which is the same as ttyS2). You may need to use `setserial` to tell Linux the IRQ you are using.

13.8 "Cannot open /dev/ttyS?: Permission denied"

Check the file permissions on this port with "ls −l /dev/ttyS?"_ If you own the ttyS? then you need read and write permissions: crw with the c (Character device) in col. 1. It you don't own it then it should show rw− in cols. 8 & 9 which means that everyone has read and write permission on it. Use "chmod" to change permissions. There are more complicated ways to get access like belonging to a "group" that has group permission.

13.9 "Operation not supported by device" for ttyS?

This means that an operation requested by setserial, stty, etc. couldn't be done because the kernel doesn't support doing it. Formerly this was often due to the "serial" module not being loaded. But with the advent of PnP, it may likely mean that there is no modem (or other serial device) at the address where the driver (and setserial) thinks it is. If there is no modem there, commands (for operations) sent to that address obviously don't get done. See What is set in my serial port hardware?

If the "serial" module wasn't loaded but "lsmod" shows you it's now loaded it might be the case that it's loaded now but wasn't loaded when you got the error message. In many cases the module will automatically loaded when needed (if it can be found). To force loading of the "serial" module it may be listed in the file: /etc/modules.conf or /etc/modules. The actual module should reside in: /lib/modules/.../misc/serial.o.

13.10 "Cannot create lockfile. Sorry"

When a port is "opened" by a program a lockfile is created in /var/lock/. Wrong permissions for the lock directory will not allow a lockfile to be created there. Use "ls −ld /var/lock" to see if the permissions are OK: usually rwx for everyone (repeated 3 times). If it's wrong, use "chmod" to fix it. Of course, if there is no "lock" directory no lockfile can be created there. For more info on lockfiles see What Are Lock Files

13.11 "Device /dev/ttyS? is locked."

This means that someone else (or some other process) is supposedly using the serial port. There are various ways to try to find out what process is "using" it. One way is to look at the contents of the lockfile (/var/lock/LCK...). It should be the process id. If the process id is say 100 type "ps 100" to find out what it is. Then if the process is no longer needed, it may be gracefully killed by "kill 100". If it refuses to be killed use "kill −9 100" to force it to be killed, but then the lockfile will not be removed and you'll need to delete it manually. Of course if there is no such process as 100 then you may just remove the lockfile but in most cases the lockfile should have been automatically removed if it contained a stale process id (such as 100).

13.12 "/dev/tty? Device or resource busy"

This means that the device you are trying to access (or use) is supposedly busy (in use) or that a resource it needs (such as an IRQ) is supposedly being used by another device (the resource is "busy"). This message is easy to understand if it only means that the device is busy (in use). But it often means that a resource is in use. What makes it even more confusing is that in some cases neither the device not the resources that it needs are actually "busy".

The ``resource busy'' part often means (example for `ttyS2`) ``You can't use `ttyS2` since another device is using ttyS2's interrupt." The potential interrupt conflict is inferred from what "setserial" thinks. A more accurate error message would be ``Can't use `ttyS2` since the setserial data (and kernel data) indicates that another device is using `ttyS2`'s interrupt". If two devices use the same IRQ and you start up only one of the devices, everything is OK because there is no conflict yet. But when you next try to start the second device (without quitting the first device) you get a "... busy" error message. This is because the kernel only keeps track of what IRQs are actually in use and actual conflicts don't happen unless the devices are in use (open). The situation for I/O address (such as 0x3f8) conflict is similar.

This error is sometimes due to having two serial drivers: one a module and the other compiled into the kernel. Both drivers try to grab the same resources and one driver finds them "busy".

There are two possible cases when you see this message:

1. There may be a real resource conflict that is being avoided.
2. Setserial has it wrong and the only reason `ttyS2` can't be used is that setserial erroneously predicts a conflict.

What you need to do is to find the interrupt setserial thinks `ttyS2` is using. Look at /proc/tty/driver/serial (if you have it). You should also be able to find it with the "setserial" command for `ttyS2`. But due to a bug (reported by me in Nov. 2000) you get the same "... busy" error message when you try this with "setserial".

To try to resolve this problem reboot or: exit or gracefully kill all likely conflicting processes. If you reboot: 1. Watch the boot–time messages for the serial ports. 2. Hope that the file that runs "setserial" at boot–time doesn't (by itself) create the same conflict again.

If you think you know what IRQ say `ttyS2` is using then you may look at /proc/interrupts to find what else (besides another serial port) is currently using this IRQ. You might also want to double check that any suspicious IRQs shown here (and by "setserial") are correct (the same as set in the hardware). A way to test whether or not it's a potential interrupt conflict is to set the IRQ to 0 (polling) using "setserial". Then if the busy message goes away, it was likely a potential interrupt conflcit. It's not a good idea to leave it permanently set at 0 since it will make the CPU work too hard.

13.13 "Input/output error" from setserial or stty

You may have typed "ttys" instead of "ttyS". You will see this error message if you try to use the setserial command for any device that is not a serial port. It also may mean that the serial port is in use (busy or opened) and thus the attempt to get/set parameters by setserial or stty failed. It could also mean that there isn't any serial port at the IO address that setserial thinks your port is at.

13.14 Overrun errors on serial port

This is an overrun of the hardware FIFO buffer and you can't increase its size. See

13.15 Troubleshooting Tools

These are some of the programs you might want to use in troubleshooting:

- "lsof /dev/ttyS*" will list serial ports which are open.
- "setserial" shows and sets the low−level hardware configuration of a port (what the driver thinks it is). See What is Setserial
- "stty" shows and sets the configuration of a port (except for that handled by "setserial"). See the section Stty
- "modemstat" or "statserial" will show the current state of various modem signal lines (such as DTR, CTS, etc.)
- "irqtune" will give serial port interrupts higher priority to improve performance.
- "hdparm" for hard−disk tuning may help some more.
- "lspci" shows the actual IRQs, etc. of hardware on the PCI bus.
- "pnpdump −−dumpregs" shows the actual IRQs, etc. of hardware for PnP devices on the ISA bus.
- Some "files" in the /proc tree (such as ioports, interrupts, and tty/driver/serial).

---

14. Interrupt Problem Details

While the section Troubleshooting lists problems by symptom, this section explains what will happen if interrupts are set incorrectly. This section helps you understand what caused the symptom, what other symptoms might be due to the same problem, and what to do about it.

14.1 Types of interrupt problems

The "setserial" program will show you how serial driver thinks the interrupts are set. If the serial driver (and setserial) has it right then everything regarding interrupts should be OK. Of course a /dev/ttyS must exist for the device and Plug−and−Play (or jumpers) must have set an address and IRQ in the hardware. Linux will not knowingly permit an interrupt conflict and you will get a "Device or resource busy" error message if you attempt to do something that would create a conflict.

Since the kernel tries to avoid interrupt conflicts and gives you the "resource busy" message if you try to create a conflict, how can interrupt conflicts happen? Easy. "setserial" may have it wrong and erroneously predicts no conflict when there will actually be a real conflict based on what is set in the hardware. When this happens there will be no "... busy" message but a conflict will physically happen. Performance is likely to be extremely slow. Both devices will send identical interrupt signals on the same wire and the CPU will erroneously think that the interrupts only come from one device. This will be explained in detail in the following sections.

Linux doesn't complain when you assign two devices the same IRQ provided that neither device is in use. As each device starts up (initializes), it asks Linux for permission to use its hardware interrupt. Linux keeps track of which interrupt is assigned to whom, and if your interrupt is already in use, you'll see this "... busy" error message. Thus if two devices use the same IRQ and you start up only one of the devices, everything is OK. But when you next try to start the second device (without quitting the first device) you get "... busy" error message.

14.2 Symptoms of Mis−set or Conflicting Interrupts

The symptoms depend on whether or not you have a modern serial port with FIFO buffers or an obsolete serial port without FIFO buffers. It's important to understand the symptoms for the obsolete ones also since sometimes modern ports seem to behave that way.

For the obsolete serial ports, only one character gets thru every several seconds. This is so slow that it seems almost like nothing is working (especially if the character that gets thru is invisible (such a space or newline). For the modern ports with FIFO buffers you will likely see bursts of up to 16 characters every several seconds.

If you have a modem on the port and dial a number, it seemingly may not connect since the CONNECT message may not make it thru. But after a long wait it may finally connect and you may see part of a login message (or the like). The response from your side of the connection may be so delayed that the other side gives up and disconnects you, resulting in a NO CARRIER message.

If you use minicom, a common test to see if things are working is to type the simplest "AT" command and see if the modem responds. Typing just at<enter> should normally (if interrupts are OK) result in an immediate "OK" response from the modem. With bad interrupts you type at<enter> and may see nothing. But then after 10 seconds or so you see the cursor drop down one line. What is going on is that the FIFO is behaving like it can only hold one byte. The "at" you typed caused it to overrun and both letters were lost. But the final <enter> eventually got thru and you "see" this invisible character by noticing that the cursor jumped down one line. If you were to type a single letter and then wait about 10 seconds, you should see it echo back to the screen. This is fine if your typing speed is less that one word per minute :−)

14.3 Mis−set Interrupts

If you don't understand what an interrupt does see Interrupts. If a serial port has one IRQ set in the hardware but a different one set in the device driver, the device driver will not catch any interrupts sent by the serial port. Since the serial port uses interrupts to call its driver to service the port (fetching bytes from its 16−byte receive buffer or putting another 16−bytes in its transmit buffer) one might expect that the serial port would not work at all.

But it still may work anyway −−sort of. Why? Well, besides the interrupt method of servicing the port there's a slow polling method that doesn't need interrupts. The way it works is that every so often the device driver checks the serial port to see if it needs anything such as if it has some bytes that need fetching from its receive buffer. If interrupts don't work, the serial driver falls back to this polling method. But this polling method was not intended to be used a substitute for interrupts. It's so slow that it's not practical to use and may cause buffer overruns. Its purpose may have been to get things going again if just one interrupt is lost or fails to do the right thing. It's also useful in showing you that interrupts have failed. Don't confuse this slow polling method with the fast polling method that operates on ports that have their IRQs set to 0.

For the 16−byte transmit buffer, 16 bytes will be transmitted and then it will wait until the next polling takes place (several seconds later) before the next 16 bytes are sent out. Thus transmission is very slow and in small chunks. Receiving is slow too since bytes that are received by the receive buffer are likely to remain there for several seconds until it is polled.

This explains why it takes so long before you see what you typed. When you type say AT to a modem, the AT goes out the serial port to the modem. The modem then echos the AT back thru the serial port to the screen. Thus the AT characters have to pass twice thru the serial port. Normally this happens so fast that AT seems to appear on the screen at the same time you hit the keys on the keyboard. With slow polling delays at the serial port, you don't see what you typed until many seconds later.

What about overruns of the 16−byte receive buffer? This will happen with an external modem since the modem just sends to the serial port at high speed which is likely to overrun the 16−byte buffer. But for an internal modem, the serial port is on the same card and it's likely to check that this receive buffer has room for more bytes before putting received bytes into it. In this case there will be no overrun of this receive buffer, but text will just appear on your screen in 16−byte chunks spaced at intervals of several seconds.

Even with an external modem you might not get overruns. If just a few characters (under 16) are sent you don't get overruns since the buffer likely has room for them. But attempts to send a larger number of bytes from your modem to your screen may result in overruns. However, more than 16 (with no gaps) can get thru without overruns if the timing is right. For example, suppose a burst of 32 bytes is sent into the port from the external cable. The polling might just happen after the first 16 bytes came in so it would pick up these 16 bytes OK. Then there would be space for the next 16 bytes so that entire 32 bytes gets thru OK. While this scenario is not very likely, similar cases where 17 to 31 bytes make thru are more likely. But it's even more likely that only an occasional 16−byte chunk will get thru with possible loss of data.

If you have an obsolete serial port with only a 1−byte buffer (or it's been incorrectly set to work like a 1−byte buffer) then the situation will be much worse than described above and only one character will occasionally make it thru the port. Every character received causes an overrun (and is lost) except for the last character received. This character is likely to be just a line−feed since this is often the last character to be transmitted in a burst of characters sent to your screen. Thus you may type AT<return> to the modem but never see AT on the screen. All you see several seconds later is that the cursor drops down one line (a line feed). This has happened to me with a 16−byte FIFO buffer that was behaving like a 1−byte buffer.

When a communication program starts up, it expects interrupts to be working. It's not geared to using this slow polling−like mode of operation. Thus all sorts of mistakes may be made such as setting up the serial port and/or modem incorrectly. It may fail to realize when a connection has been made. If a script is being used for login, it may fail (caused by timeout) due to the polling delays.

14.4 Interrupt Conflicts

When two devices have the same IRQ number it's called sharing interrupts. Under some conditions this sharing works out OK. Starting with kernel version 2.2, ISA serial ports may, if the hardware is designed for this, share interrupts with other serial ports. Devices on the PCI bus may share the same IRQ interrupt with other devices on the PCI bus (provided the software supports this). In other cases where there is potential for conflict, there should be no problem if no two devices with the same IRQ are ever "in use" at the same time. More precisely, "in use" really means "open" (in programmer jargon). In cases other than the exceptions mentioned above (unless special software and hardware permit sharing), sharing is not allowed and conflicts arise if sharing is attempted.

Even if two processes with conflicting IRQs run at the same time, one of the devices will likely have its interrupts caught by its device driver and may work OK. The other device will not have its interrupts caught by the correct driver and will likely behave just like a process with mis−set interrupts. See Mis−set Interrupts for more details.

14.5 Resolving Interrupt Problems

If you are getting a very slow response as described above, then one test is to change the IRQ to 0 (uses fast polling instead of interrupts) and see if the problem goes away. Note that the polling due to IRQ=0 is orders of magnitude faster than the slow "polling" due to bad interrupts. If IRQ=0 seems to fix the problem, then there was likely something wrong with the interrupts. Using IRQ=0 is very resource intensive and is only a temporary fix. You should try to find the cause of the interrupt problem and not permanently use IRQ=0.

Check /proc/interrupts to see if the IRQ is currently in use by another process. If it's in use by another serial port you could try "top" (type f and then enable the TTY display) or "ps −e" to find out which serial ports are in use. If you suspect that setserial has a wrong IRQ then see What is the current IO address and IRQ of my Serial Port ?

---

15. What Are UARTs? How Do They Affect Performance?

15.1 Introduction to UARTS

UARTs (**U**niversal **A**synchronous **R**eceiver **T**ransmitter) are serial chips on your PC motherboard (or on an internal modem card). The UART function may also be done on a chip that does other things as well. On older computers like many 486's, the chips were on the disk IO controller card. Still older computer have dedicated serial boards.

The UART's purpose is to convert bytes from the PC's parallel bus to a serial bit−stream. The cable going out of the serial port is serial and has only one wire for each direction of flow. The serial port sends out a stream of bits, one bit at a time. Conversely, the bit stream that enters the serial port via the external cable is converted to parallel bytes that the computer can understand. UARTs deal with data in byte sized pieces, which is conveniently also the size of ASCII characters.

Say you have a terminal hooked up to your PC. When you type a character, the terminal gives that character to its transmitter (also a UART). The transmitter sends that byte out onto the serial line, one bit at a time, at a specific rate. On the PC end, the receiving UART takes all the bits and rebuilds the (parallel) byte and puts it in a buffer.

Along with converting between serial and parallel, the UART does some other things as a byproduct (side effect) of its primary task. The voltage used to represent bits is also converted (changed). Extra bits (called start and stop bits) are added to each byte before it is transmitted. See the Serial−HOWTO section, ``Voltage Waveshapes" for details. Also, while the flow rate (in bytes/sec) on the parallel bus inside the computer is very high, the flow rate out the UART on the serial port side of it is much lower. The UART has a fixed set of rates (speeds) which it can use at its serial port interface.

15.2 Two Types of UARTs

There are two basic types of UARTs: dumb UARTS and FIFO UARTS. Dumb UARTs are the 8250, 16450, early 16550, and early 16650. They are obsolete but if you understand how they work it's easy to understand how the modern ones work with FIFO UARTS (late 16550, 16550A, 16c552, late 16650, 16750, and 16C950).

There is some confusion regarding 16550. Early models had a bug and worked properly only as 16450's (no FIFO). Later models with the bug fixed were named 16550A but many manufacturers did not accept the name change and continued calling it a 16550. Most all 16550's in use today are like 16550A's. Linux will report it as being a 16550A even though your hardware manual (or a label note) says it's a 16550. A similar situation exists for the 16650 (only it's worse since the manufacturer allegedly didn't admit anything was wrong). Linux will report a late 16650 as being a 16650V2. If it reports it as 16650 it is bad news and only is used as if it had a one−byte buffer.

15.3 FIFOs

To understand the differences between dumb and FIFO (First In, First Out queue discipline) first let's examine what happens when a UART has sent or received a byte. The UART itself can't do anything with the data passing thru it, it just receives and sends it. For the obsolete dumb UARTS, the CPU gets an interrupt from the serial device every time a byte has been sent or received. The CPU then moves the received byte out of the UART's buffer and into memory somewhere, or gives the UART another byte to send. The obsolete 8250 and 16450 UARTs only have a 1 byte buffer. That means, that every time 1 byte is sent or received, the CPU is interrupted. At low transfer rates, this is OK. But, at high transfer rates, the CPU gets so busy dealing with the UART, that is doesn't have time to adequately tend to other tasks. In some cases, the CPU does not get around to servicing the interrupt in time, and the byte is overwritten, because they are coming in so fast. This is called an "overrun" or "overflow".

FIFO UARTs help solve this problem. The 16550A (or 16550) FIFO chip comes with 16 byte FIFO buffers. This means that it can receive up to 14 bytes (or send 16 bytes) before it has to interrupt the CPU. Not only can it wait for more bytes, but the CPU then can transfer all (14 to 16) bytes at a time. This is a significant advantage over the obsolete UARTs, which only had 1 byte buffers. The CPU receives less interrupts, and is free to do other things. Data is rarely lost. Note that the interrupt threshold of FIFO buffers (trigger level) may be set at less than 14. 1, 4 and 8 are other possible choices. As of late 2000 there was no way the Linux user could set these directly (setserial can't do it). While many PC's only have a 16550 with 16−byte buffers, better UARTS have even larger buffers.

Note that the interrupt is issued slightly before the buffer gets full (at say a "trigger level" of 14 bytes for a 16−byte buffer). This allows room for a couple more bytes to be received before the interrupt service routine is able to actually fetch all these bytes. The trigger level may be set to various permitted values by kernel software. A trigger level of 1 will be almost like an obsolete UART (except that it still has room for 15 more bytes after it issues the interrupt).

Now consider the case where you're on the Internet. It's just sent you a short webpage of text. All of this came in thru the serial port. If you had a 16−byte buffer on the serial port which held back characters until it had 14 of them, some of the last several characters on the screen might be missing as the FIFO buffer waited to get the 14th character. But the 14th character doesn't arrive since you've been sent the entire page (over the phone line) and there are no more characters to send to you. It could be that these last characters are part of the HTML formatting, etc. and are not characters to display on the screen but you don't want to lose format either.

There is a "timeout" to prevent the above problem. The "timeout" works like this for the receive UART buffer: If characters arrive one after another, then an interrupt is issued only when say the 14th character reaches the buffer. But if a character arrives and the next character doesn't arrive soon thereafter, then an interrupt is issued anyway. This results in fetching all of the characters in the FIFO buffer, even if only a few (or only one) are present. There is also "timeout" for the transmit buffer as well.

15.4 UART Model Numbers

Here's a list of UARTs. *TL* is *T*rigger *L*evel

- 8250, 16450, early 16550: Obsolete with 1−byte buffers
- 16550, 16550A, 16c552: 16−byte buffers, TL=1,4,8,14
- 16650: 32−byte buffers. Speed up to 460.8 kbps
- 16750: 64−byte buffer for send, 56−byte for receive. Speed up to 921.6 kbps
- Hayes ESP: 1k−byte buffers.

The obsolete ones are only good for modems no higher than 14.4k (DTE speeds up to 38400 bps). For modern modems you need at least a 16550 (and not an early 16550). For V.90 56k modems, it may be a several percent faster with a 16650 (especially if you are downloading large uncompressed files). The main advantage of the 16650 is its larger buffer size as the extra speed isn't needed unless the modem compression ratio is high. Some 56k internal modems may come with a 16650 ??

Non−UART, and intelligent multiport boards use DSP chips to do additional buffering and control, thus relieving the CPU even more. For example, the Cyclades Cyclom, and Stallion EasyIO boards use a Cirrus Logic CD1400 RISC UART, and many boards use 80186 CPUs or even special RISC CPUs, to handle the serial IO.

Many 486 PCs (old) and all Pentiums (or the like) should have 16550As (usually called just 16550's) with FIFOs. Some better motherboards today (2000) even have 16650s. For replacing obsolete UARTs with newer ones in pre 1990 hardware see the Appendix: Obsolete ...

---

16. Pinout and Signals

16.1 Pinout

```
        PINOUT of the SERIAL PORT    (--> direction is out of PC)
               (Note DCD is sometimes labeled CD)
Pin #   Pin #   Acronym  Full-Name   Direction  What-it-May-Do/Mean
9-pin   25-pin
 3       2      TxD     Transmit Data     -->   Transmits bytes out of PC
 2       3      RxD     Receive Data      <--        Receives bytes into PC
 7       4      RTS     Request To Send   -->   RTS/CTS flow control
 8       5      CTS     Clear To Send     <--        RTS/CTS flow control
 6       6      DSR     Data Set Ready    <--        I'm ready to communicate
 4      20      DTR     Data Terminal Ready-->  I'm ready to communicate
 1       8      DCD     Data Carrier Detect<--       Modem connected to another
 9      22      RI      Ring Indicator    <--        Telephone line ringing
 5       7              Signal Ground
```

16.2 Signals May Have No Fixed Meaning

Only 3 of the 9 pins have a fixed assignment: transmit, receive and signal ground. This is fixed by the hardware and you can't change it. But the other signal lines are controlled by software and may do (and mean) almost anything at all. However they can only be in one of two states: asserted (+12 volts) or negated (−12 volts). Asserted is "on" and negated is "off". For example, Linux software may command that DTR be negated and the hardware only carries out this command and puts −12 volts on the DTR pin. A modem (or other device) that receives this DTR signal may do various things. If a modem has been configured a certain way it will hang−up the telephone line when DTR is negated. In other cases it may ignore this signal or do something else when DTR is negated (turned off).

It's like this for all the 6 signal lines. The hardware only sends and receives the signals, but what action (if any) they perform is up to the Linux software and the configuration/design of devices that you connect to the serial port. However, most pins have certain functions which they normally perform but this may vary with the operating system and the device driver configuration. Under Linux, one may modify the source code to make these signal lines behave differently (some people have).

16.3 Cabling Between Serial Ports

A cable from a serial port always connects to another serial port. A modem or other device that connects to the serial port has a serial port built into it. For modems, the cable is always straight thru: pin 2 goes to pin 2, etc. The modem is said to be DCE (Data Communications Equipment) and the computer is said to be DTE (Data Terminal Equipment). Thus for connecting DTE−to−DCE you use straight−thru cable. For connecting DTE−to−DTE you must use a null−modem cable and there are many ways to wire such cable (see examples in Text−Terminal−HOWTO subsection: "Direct Cable Connection")

There are good reasons why it works this way. One reason is that the signals are unidirectional. If pin 2 sends a signal out of it (but is unable to receive any signal) then obviously you can't connect it to pin 2 of the same type of device. If you did, they would both send out signals on the same wire to each other but neither would be able to receive any signal. There are two ways to deal with this situation. One way is to have a two different types of equipment where pin 2 of the first type sends the signal to pin 2 of the second type (which receives the signal). That's the way it's done when you connect a PC (DTE) to a modem (DCE). There's a second way to do this without having two different types of equipment: Connect pin sending pin 2 to a receiving pin 3 on same type of equipment. That's the way it's done when you connect 2 PC's together or a PC to a terminal (DTE−to−DTE). The cable used for this is called a null−modem cable. The above example is for a 25 pin connector but for a 9−pin connector the pin numbers are just the opposite.

The serial pin designations were originally intended for connecting a dumb terminal to a modem. The terminal was DTE (Data Terminal Equipment) and the modem was DCE (Data Communication Equipment). Today the PC is usually used as DTE instead of a terminal (but real terminals may still be used this way). The names of the pins are the same on both DTE and DCE. The words: "receive" and "transmit" are in this case from the point of view of the PC (DTE). The transmit pin from the PC transmits to the "transmit" pin of the modem (but actually the modem is receiving the data from this pin so from the point of view of the modem it would be a receive pin).

The serial port was originally intended to be used for connecting DTE to DCE which makes cabling simple: just use a straight−thru cable. Thus when one connects a modem one seldom needs to worry about which pin is which. But people wanted to connect DTE to DTE (for example a computer to a terminal) and various ways were found to do this by fabricating various type of special cables. In this case what pin connects to what pin becomes more important.

16.4 RTS/CTS and DTR/DSR Flow Control

This is "hardware" flow control. Flow control was previously explained in the Flow Control subsection but the pins and voltage signals were not. Linux only supports RTS/CTS flow control at present (but a special driver may exist for a specific application which supports DTR/DSR flow control). Only RTS/CTS flow control will be discussed since DTR/DSR flow control works the same way. To get RTS/CTS flow control

one needs to either select hardware flow control in an application program or use the command:
stty crtscts < /dev/ttyS2 (or the like). This enables RTS/CTS hardware flow control in the Linux device driver.

Then when a DTE (such as a PC) wants to stop the flow into it, it negates RTS. Negated "Request To Send" (−12 volts) means "request NOT to send to me" (stop sending). When the PC is ready for more bytes it asserts RTS (+12 volts) and the flow of bytes to it resumes. Flow control signals are always sent in a direction opposite to the flow of bytes that is being controlled. DCE equipment (modems) works the same way but sends the stop signal out the CTS pin. Thus it's RTS/CTS flow control using 2 lines.

On what pins is this stop signal received? That depends on whether we have a DCE−DTE connection or a DTE−DTE connection. For DCE−DTE it's a straight−thru connection so obviously the signal is received on a pin with the same name as the pin it's sent out from. It's RTS−−>RTS (PC to modem) and CTS<−−CTS (modem to PC). For DTE−to−DTE the connection is also easy to figure out. The RTS pin always sends and the CTS pin always receives. Assume that we connect two PCs (PC1 and PC2) together via their serial ports. Then it's RTS(PC1)−−>CTS(PC2) and CTS(PC1)<−−RTS(PC2). In other words RTS and CTS cross over. Such a cable (with other signals crossed over as well) is called a "null modem" cable. See Cabling Between Serial Ports

What is sometimes confusing is that there is the original use of RTS where it means about the opposite of the previous explanation above. This original meaning is: I Request To Send to you. This request was intended to be sent from a terminal (or computer) to a modem which, if it decided to grant the request, would send back an asserted CTS from its CTS pin to the CTS pin of the computer: You are Cleared To Send to me. Note that in contrast to the modern RTS/CTS bi−directional flow control, this only protects the flow in one direction: from the computer (or terminal) to the modem. This original use appears to be little used today on modern equipment (including modems).

The DTR and DSR Pins

Just like RTS and CTS, these pins are paired. For DTE−to−DTE connections they are likely to cross over. There are two ways to use these pins. One way is to use them as a substitute for RTS/CTS flow control. The DTR pin is just like the RTS pin while the DSR pin behaves like the CTS pin. Although Linux doesn't support DTR/DSR flow control, it can be obtained by connecting the RTS/CTS pins at the PC to the DSR/DTR pins at the device that uses DTR/DSR flow control. DTR flow control is the same as DTR/DSR flow control but it's only one−way and only uses the DTR pin at the device. Many text terminals and some printers use DTR/DSR (or just DTR) flow control. In the future, Linux may support DTR/DSR flow control. The software has already been written but it's not clear when (or if) it will incorporated into the serial driver.

The normal use of DTR and DSR (not for flow control) is as follows: A device asserting DTR says that its powered on and ready to operate. For a modem, the meaning of a DTR signal from the PC depends on how the modem is configured. Sending a DTR signal manually from a PC using the stty command sets the baud rate to 0. Negating DTR is sometimes called "hanging up" but it doesn't always do this.

16.5 Preventing a Port From Opening

If "stty −clocal" (or getty is used with the "local" flag negated) then a serial port can't open until DCD gets an assert (+12 volts) signal.

17. Voltage Waveshapes

17.1 Voltage for a Bit

At the EIA−232 serial port, voltages are bipolar (positive or negative with respect to ground) and should be about 12 volts in magnitude (some are 5 or 10 volts). For the transmit and receive pins +12 volts is a 0−bit (sometimes called "space") and −12 volts is a 1−bit (sometimes called "mark"). This is known as inverted logic since normally a 0−bit is both false and negative while a one is normally both true and positive. Although the receive and transmit pins are inverted logic, other pins (modem control lines) are normal logic with a positive voltage being true (or "on" or "asserted") and a negative voltage being false (or "off" or "negated"). Zero voltage has no meaning (except it usually means that the unit is powered off).

A range of voltages is allowed. The specs say the magnitude of a transmitted signal should be between 5 and 15 volts but must never exceed 25 V. Any voltage received under 3 V is undefined (but some devices will accept a lower voltage as valid). One sometimes sees erroneous claims that the voltage is commonly 5 volts (or even 3 volts) but it's usually 11−12 volts. If you are using a EIA−422 port on a Mac computer as an EIA−232 (requires a special cable) or EIA−423 then the voltage will actually be only 5 V. The discussion here assumes 12 V.

Note that normal computer logic normally is just a few volts (5 volts was once the standard) so that if you try to use test equipment designed for testing 3−5 volt computer logic (TTL) on the 12 volts of a serial port, it may damage the test equipment.

17.2 Voltage Sequence for a Byte

The transmit pin (TxD) is held at −12 V (mark) at idle when nothing is being sent. To start a byte it jumps to +12 V (space) for the start bit and remains at +12 V for the duration (period) of the start bit. Next comes the low−order bit of the data byte. If it's a 0−bit nothing changes and the line remains at +12 V for another bit−period. If it's a 1−bit the voltage jumps from +12 to −12 V. After that comes the next bit (−12 V if a 1 or +12 V if a 0), etc., etc. After the last data bit a parity bit may be sent and then a −12 V (mark) stop bit. Then the line remains at −12 V (idle) until the next start bit. Note that there is no return to 0 volts and thus there is no simple way (except by a synchronizing signal) to tell where one bit ends and the next one begins for the case where 2 consecutive bits are the same polarity (both zero or both one).

A 2nd stop bit would also be −12 V, just the same as the first stop bit. Since there is no signal to mark the boundaries between these bits, the only effect of the 2nd stop bit is that the line must remain at −12 V idle twice as long. The receiver has no way of detecting the difference between a 2nd stop bit and a longer idle time between bytes. Thus communications works OK if one end uses one stop bit and the other end uses 2 stop bits, but using only one stop bit is obviously faster. In rare cases 1 1/2 stop bits are used. This means that the line is kept at −12 V for 1 1/2 time periods (like a stop bit 50% wider than normal).

17.3 Parity Explained

Characters are normally transmitted with either 7 or 8 bits of data. An additional parity bit may (or may not) be appended to this resulting in a byte length of 7, 8 or 9 bits. Some terminal emulators and older terminals do not allow 9 bits. Some prohibit 9 bits if 2 stop bits are used (since this would make the total number of bits too large: 12 bits total after adding the start bit).

The parity may be set to odd, even or none (mark and space parity may be options on some terminals or other serial devices). With odd parity, the parity bit is selected so that the number of 1−bits in a byte, including the parity bit, is odd. If a such a byte gets corrupted by a bit being flipped, the result is an illegal byte of even parity. This error will be detected and if it's an incoming byte to the terminal an error−character symbol will appear on the screen. Even parity works in a similar manner with all legal bytes (including the parity bit) having an even number of 1−bits. During set−up, the number of bits per character usually means only the number of data bits per byte (7 for true ASCII and 8 for various ISO character sets).

A "mark" is a 1−bit (or logic 1) and a "space" is a 0−bit (or logic 0). For mark parity, the parity bit is always a one−bit. For space parity it's always a zero−bit. Mark or space parity (also known as "sticky parity") only wastes bandwidth and should be avoided if feasible. The `stty` command can't set sticky parity but it's supported by serial hardware and can be dealt with by programming in C. "No parity" means that no parity bit is added. For terminals that don't permit 9 bit bytes, "no parity" must be selected when using 8 bit character sets since there is no room for a parity bit.

17.4 Forming a Byte (Framing)

In serial transmission of bytes via EIA−232 ports, the low−order bit is always sent first. Serial ports on PC's use asynchronous communication where there is a start bit and a stop bit to mark the beginning and end of a byte. This is called framing and the framed byte is sometimes called a frame. As a result a total of 9, 10, or 11 bits are sent per byte with 10 being the most common. 8−N−1 means 8 data bits, No parity, 1 stop bit. This adds up to 10 bits total when one counts the start bit. One stop bit is almost universally used. At 110 bits/sec (and sometimes at 300 bits/sec) 2 stop bits were once used but today the 2nd stop bit is used only in very unusual situations (or by mistake since it still works OK that way but wastes bandwidth).

Don't confuse this type of framing with the framing used for a packet of bytes on a network. The serial port just frames every byte. For a network many bytes are framed into a packet (sometimes called a frame). For a network frame, instead of a start bit, there is a sequence of bytes called a header. On a network that uses serial ports (with modems), a report of a frame error usually refers to a multi−byte frame and not the serial port frame of a single byte.

17.5 How "Asynchronous" is Synchronized

The EIA−232 serial port as implemented on PC is asynchronous which in effect means that there is no "clock" signal sent with "ticks" to mark when each bit is sent.. There are only two states of the transmit (or receive) wire: mark (−12 V) or space (+12 V). There is no state of 0 V. Thus a sequence of 1−bits is transmitted by just a steady −12 V with no markers of any kind between bits. For the receiver to detect individual bits it must always have a clock signal which is in synchronization with the transmitter clock. Such a clock would generate a "tick" in synchronization with each transmitted (or received) bit.

For asynchronous transmission, synchronization is achieved by framing each byte with a start bit and a stop bit (done by hardware). The receiver listens on the negative line for a positive start bit and when it detects one it starts its clock ticking. It uses this clock tick to time the reading of the next 7, 8 or 9 bits. (It actually is a little more complex than this since several samples of a bit are normally taken and this requires additional timing ticks.) Then the stop bit is read, the clock stops and the receiver waits for the next start bit. Thus async is actually synchronized during the reception of a single byte but there is no synchronization between one byte and the next byte.

18. Other Serial Devices (not async EIA−232)

18.1 Successors to EIA−232

A number of EIA standards have been established for higher speeds and longer distances using twisted−pair (balanced) technology. Balanced transmission can sometimes be a hundred times faster than unbalanced EIA−232. For a given speed, the distance (maximum cable length) may be many times longer with twisted pair. But PC−s keep being made with the "obsolete" EIA−232 since it works OK with modems and mice since the cable length is short. If this appears in the latest version of this HOWTO, please let me know if any of the non−EIA−232 listed below are supported by Linux.

18.2 EIA−422−A (balanced) and EIA−423−A (unbalanced)

EIA−423 is just like the unbalanced EIA−232 except that the voltage is only 5 volts. Since this falls within EIA−232 specs it can be connected to a EIA−232 port. Its specs call for somewhat higher speeds than the EIA−232 (but this may be of little help on a long run where it's the unbalance that causes interference). Since EIA−423 is not much of an improvement over EIA−232, it is not popular except on old Mac computers.

EIA−422 is twisted pair (known as "balanced" or "differential) and is (per specs) exactly 100 times as fast as EIA−423 (which in turn is somewhat faster than EIA−232). Apple's Mac computer prior to mid−1998 with its EIA−232/EIA−422 Port uses it. The Mac used a small round "mini−DIN−8" connector. It also provided conventional EIA−232 but at only at 5 volts (which is still legal EIA−232). To make it work like at EIA−232 one must use a special cable which (signal) grounds RxD+ (one side of a balanced pair) and use RxD− as the receive pin. While TxD− is used as the transmit pin, for some reason TxD+ should not be grounded. See Macintosh Communications FAQ. However, due to the fact that Macs (and upgrades for them) cost more than PC's, they are not widely as host computers for Linux.

18.3 EIA−485

This is like EIA−422 (balanced = differential). It is half−duplex. It's not just point−to−point but is like ethernet or the USB since all devices (nodes) on it share the same "bus". It may be used for a multidrop LAN (up to 32 nodes or more). Since many nodes share the same twisted pair the need to use the electrical tri−state mode where besides the 0 and 1 binary states there is also an open circuit state to permit other nodes to uses the twisted pair line. Instead of a transmitter keeping a 1−state voltage on the line during line idle, the line is open circuited and all nodes just listen (receive mode).

The most common architecture is master/slave. The master polls the slaves to see if they have anything to send. A slave can only transmit just after it's been polled.

There is an alternative implementation where two pair of wires are used for sending data. One pair is only for the Master to send to the Slaves. Since no one transmits on this line except the master, there is no need for it to be tri−state. Thus the Master may just be EIA−232 but the slaves must still be EIA−485. See http://www.hw.cz/english/docs/rs485/rs485.html for more details.

18.4 EIA−530

EIA−530−A (balanced but can also be used unbalanced) at 2Mbits/s (balanced) was intended to be a replacement for EIA−232 but few have been installed. It uses the same 25−pin connector as EIA−232.

18.5 EIA−612/613

The High Speed Serial Interface (HSSI = EIA−612/613) uses a 50−pin connector and goes up to about 50 Mbits/s but the distance is limited to only several meters. For Linux there are PCI cards supporting HSSI. The companies that sell the cards often provide (or point you to) a Linux driver. A howto or the like is needed for this topic.

18.6 The Universal Serial Bus (USB)

The Universal Serial Bus (USB) is being built into PCI chips. Newer PC's have them. It is 12 Mbps (with 200 Mbps planned) over a twisted pair with a 4−pin connector (2 wires are power supply). It also is limited to short distances of at most 5 meters (depends on configuration). Linux supports the bus, although not all devices that can plug into the bus are supported.

It is synchronous and transmits in special packets like a network. Just like a network, it can have several devices attached to it. Each device on it gets a time−slice of exclusive use for a short time. A device can also be guaranteed the use of the bus at fixed intervals. One device can monopolize it if no other device wants to use it. It's not simple to describe in detail.

For documentation, see the USB directory in /usr/share/doc/kernel ... It would be nice to have a HOWTO on the USB. See also http://www.linux−usb.org and/or http://.www.qbik.ch/usb/.

18.7 Firewire

Firewire (IEEE 1394) is something like the USB only faster (800 Mbps is planned). The protocol on the bus is claimed to be more efficient than USB's. It uses two twisted pair for data plus two power conductors (6 conductors in all). A variants uses only 4 conductors. You may compile firewire support into the Linux kernel. Like USB, it's also limited to short distances.

18.8 Synchronization & Synchronous

Beside the asynchronous EIA−232 (and others) there are a number of synchronous serial port standards. In fact EIA−232 includes synchronous specifications but they aren't normally implemented for serial ports on PC's. But first we'll explain what a synchronous means.

Defining Asynchronous vs Synchronous

Asynchronous (async) means "not synchronous". In practice, an async signal is what the async serial port sends and receives which is a stream of bytes each delimited by a start and stop bit. Synchronous (sync) is most everything else. But this doesn't explain the basic concepts.

In theory, synchronous means that bytes are sent out at a constant rate one after another in step with a clock signal tick. There is often a separate wire or channel for sending the clock signal. Asynchronous bytes may be sent out erratically with various time intervals between bytes (like someone typing characters at a keyboard).

There are certain situations that need to be classified as either sync or async. The async serial port often sends out bytes in a steady stream which would make this a synchronous case but since they still have the start/stop bits (which makes it possible to send them out erratically) it's called async. Another case is where data bytes (without any start−stop bits) are put into packets with possible erratic spacing between one packet and the next. This is called sync since the bytes within each packet must be transmitted synchronously.

Synchronous Communication

Did you ever wonder what all the unused pins are for on a 25−pin connector for the serial port? Most of them are for use in synchronous communication which is seldom implemented on PC's. There are pins for sync timing signals as well as for a sync reverse channel. The EIA−232 spec provides for both sync and async but PC's use a UART (Universal Asynchronous Receiver/Transmitter) chip such as a 16450, 16550A, or 16650 and can't deal with sync. For sync one needs a USART chip or the equivalent where the "S" stands for Synchronous. Since sync is a niche market, a sync serial port is likely to be quite expensive.

Besides the sync part of the EIA−232, there are various other EIA synchronous standards. For EIA−232, 3 pins of the connector are reserved for clock (or timing) signals. Sometimes it's a modem's task to generate some timing signals making it impossible to use synchronous communications without a synchronous modem (or without a device called a "synchronous modem eliminator" which provides the timing signals).

Although few serial ports are sync, synchronous communication does often take place over telephone lines using modems which use V.42 error correction. This strips off the start/stop bits and puts the date bytes in packets resulting in synchronous operation over the phone line.

---

19. Other Sources of Information

19.1 Books

1. Axleson, Jan: Serial Port Complete, Lakeview Research, Madison, WI, 1998.
2. Black, Uyless D.: Physical Layer Interfaces & Protocols, IEEE Computer Society Press, Los Alamitos, CA, 1996.
3. Campbell, Joe: The RS−232 Solution, 2nd ed., Sybex, 1982.
4. Campbell, Joe: C Programmer's Guide to Serial Communications, 2nd ed., Unknown Publisher, 1993.
5. Levine, Donald: POSIX Programmer's Guide, O'Reilly, 1991.
6. Nelson, Mark: Serial Communications Developer's Guide, 2nd ed., Hungry Minds, 2000.
7. Putnam, Byron W.: RS−232 Simplified, Prentice Hall, 1987.
8. Seyer, Martin D.: RS−232 Made Easy, 2nd ed., Prentice Hall, 1991.
9. Stevens, Richard W.: Advanced Programming in the UNIX Environment, (ISBN 0−201−56317−7; Addison−Wesley)
10. Tischert, Michael & Bruno Jennrich: PC Intern, Abacus 1996. Chapter 7: Serial Ports

Notes re books:

1. "... Complete" has hardware details (including register) but the programming aspect is Window oriented.
2. "Physical Layer ..." covers much more than just EIA−232.

19.2 Serial Software

It's best to use the nearest mirror site, but here's the main sites:
Serial Software for Linux software for the serial ports including getty and port monitors.

Serial Communications for communication programs.

- `irqtune` will give serial port interrupts higher priority to improve performance. Using `hdparm` for hard−disk tuning may help some more.
- `modemstat` and `statserial` show the current state of various modem control lines. See Serial Monitoring/Diagnostics

19.3 Related Linux Documents

- man pages for: `setserial` and `stty`
- Low−Level Terminal Interface part of "GNU C Library Reference manual" (in libc (or glibc) docs package). It covers the detailed meaning of "stty" commands, etc.
- Modem−HOWTO: modems on the serial port
- PPP−HOWTO: help with PPP (using a modem on the serial port)
- Printing−HOWTO: for setting up a serial printer
- Serial−Programming−HOWTO: for some aspects of serial−port programming
- Text−Terminal−HOWTO: how they work and how to install and configure
- UPS−HOWTO: setting up UPS sensors connected to your serial port
- UUCP−HOWTO: for information on setting up UUCP

19.4 Usenet newsgroups:

- comp.os.linux.answers
- comp.os.linux.hardware: Hardware compatibility with the Linux operating system.
- comp.os.linux.networking: Networking and communications under Linux.
- comp.os.linux.setup: Linux installation and system administration.

19.5 Serial Mailing List

The Linux serial mailing list. To join, send email to `majordomo@vger.rutgers.edu`, with ``` ``subscribe linux-serial ```" in the message body. If you send ``` ``help ```" in the message body, you get a help message. The server also serves many other Linux lists. Send the ``` ``lists ```" command for a list of mailing lists.

19.6 Internet

- Linux Serial Driver home page Includes info about PCI support.
- Serial Suite by Vern Hoxie is a collection of blurbs about the care and feeding of the Linux serial port plus some simple programs. He also has a Serial−Programming−HOWTO (not yet available from the Linux Documentation Project). Your browser should automatically log you in but if you do it manually login as "anonymous" and use your full e−mail address as the password.
- A white paper discussing serial communications and multiport serial boards was available from Cyclades at `http://www.cyclades.com`.

20. Appendix: Obsolete Hardware (prior to 1990) Info

20.1 Replacing obsolete UARTS

Many 486 PCs (old) and all Pentiums (or the like) should have modern 16550As (usually called just 16550's) with FIFOs. If you have something really old the chip may unplug so that you may be able to upgrade by buying a 16550A chip and replacing your existing 16450 UART. If the functionality has been built into another type of chip, you are out of luck. If the UART is socketed, then upgrading is easy (if you can find a replacement). The new and old are pin−to−pin compatible. It may be more feasible to just buy a new serial card on the Internet (few retail stores stock them today) or find a used one.

END OF Serial−HOWTO