Serial Programming

Wikibooks.org

March 18, 2013

On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. An URI to this license is given in the list of figures on page 141. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 139. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 145, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 141. This PDF was generated by the LaTeX typesetting software. The LaTeX source code is included as an attachment (`source.7z.txt`) in this PDF file. To extract the source from the PDF file, we recommend the use of `http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/` utility or clicking the paper clip attachment symbol on the lower left of your PDF Viewer, selecting `Save Attachment`. After extracting it from the PDF file you have to rename it to `source.7z`. To uncompress the resulting archive we recommend the use of `http://www.7-zip.org/`. The LaTeX source itself was generated by a program written by Dirk Hünniger, which is freely available under an open source license from `http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf`. This distribution also contains a configured version of the `pdflatex` compiler with all necessary packages and fonts needed to compile the LaTeX source included in this PDF file.

Contents

1 Introduction and OSI Model **3**

1.1 Introduction 3
1.2 Why Serial Communication? 3
1.3 OSI Layered Network Communications Model 4
1.4 Software Examples 5
1.5 Applications in Education 5
1.6 External Links / References 6
1.7 Other Serial Programming Articles 6

2 RS-232 Connections **7**

2.1 Introduction 7
2.2 Data Terminal/Communications Equipment 7
2.3 Connection Types 12
2.4 Wiring Pins Explained 16
2.5 Baud Rates Explained 19
2.6 Signal Bits 22
2.7 Relationship of Baud Rate to Maximum Distance 24
2.8 External References 25
2.9 Other Serial Programming Articles 26

3 8250 UART Programming **27**

3.1 Introduction 27
3.2 8086 I/O ports 28
3.3 x86 Processor Interrupts 30
3.4 8259 PIC (Programmable Interrupt Controller) 32
3.5 Serial COM Port Memory and I/O Allocation 35
3.6 UART Registers 36
3.7 Software Identification of the UART 52
3.8 External References 53
3.9 Other Serial Programming Articles 54

4 Serial DOS **55**

4.1 Introduction 55
4.2 **Hello World**, Serial Data Version 55
4.3 Finding the Port I/O Address for the UART 56
4.4 Making modifications to UART Registers 59
4.5 Basic Serial Input 60
4.6 Interrupt Drivers in DOS 63
4.7 Terminal Program Revisited 68

5 **Serial Linux** **75**
5.1 The Classic Unix C APIs for Serial Communication 75
5.2 Serial I/O on the Shell Command Line 80
5.3 System Configuration 84
5.4 Other Serial Programming Articles 85

6 **Serial Java** **87**
6.1 Using Java for Serial Communication 87
6.2 JavaComm API 90
6.3 RxTx 105
6.4 See also 106

7 **Forming Data Packets** **109**
7.1 For further reading 112

8 **Error Correction Methods** **115**
8.1 Introduction 115
8.2 ACK-NAK 115
8.3 FEC 118
8.4 Pretend It Never Happened 118
8.5 combination 119
8.6 further reading 119
8.7 further reading 119

9 **Appendex A:Modems and AT Commands** **121**
9.1 Introduction 121
9.2 Modem Programming Basics 126
9.3 Flow Control 132
9.4 Changing State 133
9.5 Sync. vs. Async. Interface 134
9.6 X.25 Interface 134
9.7 AT Commands 134
9.8 Result Codes 137
9.9 S-Registers 138
9.10 Advanced Features 138

10 **Contributors** **139**

List of Figures **141**

11 **Licenses** **145**
11.1 GNU GENERAL PUBLIC LICENSE 145
11.2 GNU Free Documentation License 146
11.3 GNU Lesser General Public License 147

1 Introduction and OSI Model

1.1 Introduction

Welcome to the wonderful world of serial data communications. This is a part of a series of articles that will cover many aspects of serial data communications. I am going to try and start from the beginning and follow a layered approach to working with serial data and by the time we are through we should be able to transfer just about any sort of data that you would care to send over wires between computers. Possibly even without wires (wireless data communication).

There are so many aspects about this subject that sometimes it is a very hard nut to crack. I'm going to dive down and try to start with the basics and introducing the RS-232 serial data communications standard.

1.2 Why Serial Communication?

First of all, the basic standards that I will be describing are, from the perspective of computer technology, positively ancient. Some of you reading this could perhaps find your grandparents or even great-grandparents using this protocol when they were in College. At the same time, it is so solid in concept that the reason for abandoning it should always be questioned. Indeed, there have been several other data transmission methods that have been developed since the RS-232 serial data protocol was established, but this workhorse is still widely used and seems to go through a rebirth every once in a while.

When all else fails, RS-232 serial communication can be relied upon. When you are trying to get two pieces of computer equipment together, sometimes newer communications methods have hard limitations that can't be worked out due to number of connections, RF interference, distance limitations, being behind physical barriers, in sensitive areas like medical equipment where stray voltages can be a problem, or that you absolutely need to rely upon the data being transmitted. A sister protocol to RS-232, the RS-422 protocol, even allows transmissions for several miles of cable.

Serial data communication is widely implemented. While it is sometimes presumed that a PC can deal with just about any problem you want to throw at it, there are a number of electronic devices that are full of data which needs to be recorded. In part because of the age of this protocol, there are many legacy devices that have RS-232 serial data as the only access to the outside world. But even many of the latest network devices have RS-232 "console" ports to facilitate initial configuration and provide a means of troubleshooting when the network itself is broken. Because the hardware is so widely implemented and available, together with many software tools, it is also relatively cheap to develop equipment

and software using this system. Particularly when transmission speed isn't important, but data needs to be sent on a regular basis. RS-232 serial data is a very reasonable solution instead of a more expensive 10BASE-T TCP/IP solution or high-speed fiber optics.

Serial data communication is also versatile. While the usual method of transmission is over copper wires between two fixed points, recently there have been some converters that transmit serial data over fiber optic lines, wireless transmitters, USB devices, and even over TCP/IP networks. What is really surprising here is that all of these transmission methods are totally transparent to the device receiving or transmitting the serial data. It can also be a carrier for TCP/IP, and be used for private networks.

1.3 OSI Layered Network Communications Model

While serial data communication is not strictly a network communication protocol, it is still important to understand the layered communications model when dealing with any sort of communications protocols. Often people implementing serial data software have to build multiple layers of this model, even if they are not totally aware of it when they are doing it at the time.

Network Layers:

- Application
- Presentation
- Session
- Transport
- Network
- Data-Link
- Physical

Often serial data communication does not implement all of these different layers, and even more often these different layers are combined in the same module or even the very same function. This model was originally developed by the International Organization for Standards (ISO) in 1984 to help give a good idea of where different networking structures could be separated and intermingled. The point here is to know that you can separate different parts of communications sub-systems to help with the debugging process, and to move structures from one sub-system to another.

If your software is well written using a model similar to this one, the software subroutines in layers above and below do not have to be rewritten if the module at a particular layer is changed. To achieve this you need to establish strong standards for the interface between the layers, which will be covered in other sections of these articles. For example, a web browser does not need to know if the HTML is being sent over fiber optic cables, wireless transmissions, or even over a serial data cable.

1.3.1 Serial Comm Layers

For serial data communication, I see this layer model as more common:

- Serial Data Applications

- Serial Networks
- Packet Challenge/Verification
- Basic Serial Packets
- 8250 UART processing
- Raw RS-232 Signals

In the case of many serial data applications, not all of these layers are implemented. Often it is just raw packets being transmitted in one direction, but sometimes even just a signal of any kind can indicate some action take place on a computer, regardless of content. It is possible to simply take the logic level of a raw RS-232 signal in your software, but at some point the data does need to be converted and the voltages involved with RS-232 can damage hardware, so this is very seldom done.

1.4 Software Examples

I don't want to get into a holy war over programming languages with this series of articles. For the moment, I'm going to be using Turbo Pascal and Delphi as the programming languages, if for no other reason then the fact that I am most comfortable programming in this development environment. If a good C/C++ guru would like to "translate" these routines, I would welcome that, as well as other programming languages where applicable. Serial communication is complicated enough so please avoid esoteric languages like Intercal or Malbolge. A good BASIC implementation would be welcome, as would LISP. I'll try to avoid language-specific features and simply deal with functions in a generic sense, which good programmers should be able to translate to the language of their choice.

These articles are meant to teach you the basics of serial data communication, not to be a functioning serial data driver. Still, all code examples will be checked and sent through an actual compiler before being listed in the articles, and hopefully fully debugged. There is no one single way to accomplish these steps and tasks, so I am going to encourage a hands-on approach to dealing with software and setting up networks.

While I've had quite a bit of experience in dealing with several serial data protocols (on the packet level), I am by no means the topmost expert at this. As I said earlier, I have considerable experience in dealing with communications at many levels, and I'd like to share some of my very hard-won knowledge.

1.5 Applications in Education

While I am only a Software Engineer and don't have the "formal" credentials necessary for making an educational textbook, I do believe that there is much that could be taught about computer networking by students experimenting with serial data communication. The audience that I am aiming for with these articles are the High School hackers/computer geeks and undergraduate CS majors. A High School teacher that wanted to tackle a subject like this, or if you wanted to cover a special topic course in a university setting where students could get some very hands-on experience with communications protocols. Every layer of

the OSI model could be demonstrated in a manner that students would learn from first-hand experiences why certain rules/systems have been implemented on the Internet, what standards documents mean, and perhaps even participate in creating standards documents.

If you are a professor or High School instructor interested in using this text, I would be particularly interested in adapting this text to better suit your needs, or working with you in covering this subject.

From a professional**perspective**, this is a topic that is seldom taught at a university, and usually only in passing when they are rushing through a whole bunch of other protocol suites. Software developers are usually introduced to this topic by having their supervisor dump a bunch of specification documents on their desk, a driver disk with API documentation, and perhaps a typically short deadline in order to get something working that should have been working sometime last year. Software developers who really understand serial data communication are worth gold, and often even these developers only learn just enough to get the immediate job done.

I've also found that skills learned from developing serial data communications also translate into other projects and give a deeper understanding of just about any data transmission system. In addition to the other groups I mentioned, I am also aiming for those unfortunate software engineers who are trying to learn just about anything about this very difficult subject and don't know where to begin. Documentation about serial communication is sparse, and sometime contradictory.

This doesn't have to be that complicated of a subject, and it is possible for mere mortals to be able to understand how everything works.

1.6 External Links / References

- Cisco explanation of the OSI model[1]
- University of Indiana / Unix Support Group explanation of OSI[2]
- ISO catalog of OSI standards[3]

1.7 Other Serial Programming Articles

Category:Serial Programming[4]

1 http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introint.htm
2 http://www.uwsg.iu.edu/usail/network/nfs/network_layers.html
3 http://www.iso.org/iso/en/CatalogueListPage.CatalogueList?ICS1=35&ICS2=100
4 http://en.wikibooks.org/wiki/Category%3ASerial%20Programming

2 RS-232 Connections

2.1 Introduction

The RS-232 standard is a collection of connection standards between different pieces of equipment. This is a rather old standard, and has been revised many times over the years to accommodate changes to communications technology. A bare-bones connection will have only one wire connected between two pieces of equipment, but usually there are more. Three wires (transmit, receive, and ground) are usually the minimum recommended. A fully implemented RS-232 connection can have as many as 25 wires between each end. Some of the early RS-232 connections were also used to connect terminal equipment to modems, so information about modems is sometimes found with general serial data communication.

2.2 Data Terminal/Communications Equipment

In the world of serial communications, there are two different kinds of equipment:

- DTE - Data Terminal Equipment
- DCE - Data Communications Equipment

2.2.1 Straight Serial Connections

In practice the distinction between the two pieces of equipment is really a matter of function rather than any real difference. As mentioned earlier, modems and serial communication equipment have been mixed together, this is another case of that. In this situation, the modem can be thought of as the Data Communications Equipment (DCE) and the terminal that somebody is sitting down and using is the Data Terminal Equipment. In the older days when it was common to use a timeshare computer system (pre 1980s), you would dial up a telephone, stick the handset that you would normally talk with into an acoustical modem, and that modem would be connected to a simple dumb terminal with an RS-232 cable. When we get to baud rates this will make more sense, but the typical connection speed was usually either 50 baud or 110 baud, and really fast connections going at 300 baud.

As a side note, when the very first IMPs (Interconnection Message Processors) that formed the first nodes/routers of ARPAnet (the ancient predecessor of the Internet), this was exactly the connection system they were using. This later gave way to other communication systems, but this was the beginning of the Internet.

In a more modern setting, imagine a piece of equipment in a very dangerous place, like in a steel processing mill that measures the temperature of the rollers or other steel processing equipment. This would also be a form of what we now refer to as a piece of "Data

Communication Equipment" that we would also want to be able to control remotely. The PC that is used in a control room of the mill would be the Data Terminal Equipment. There are many other similar kinds of devices, and RS-232 connections can be found on all kinds of equipment.

The reason this is called a "straight" connection is because when the cabling is put together, each wire on each end of the connection is put to the same pin. This wiring system will be explained further on.

2.2.2 Null Modems

Often you don't always want to connect a piece of equipment to a computer, but you would also like to connect two computers together. Unfortunately, when connecting two computers with a "straight" serial connection, the two computers are fighting each other on the same wires.

One way to make this work is to connect the two computers to each other with a pair of modems. As explained earlier, this is a very common task, and in the 1980's and early 1990's it was common to have "Bulletin Board Systems" (BBS) where computers would call each other up with modems and exchange all sorts of information.

Now imagine if these two computers are in the very same room. Instead of going through the physical modems, they go through a "null modem", or a modem that really doesn't exist. In order to make this work you have to "cross" some of the wires so when you transmit some information on one end, the other computer is able to detect and receive that same information.

In addition to simply allowing a computer to communicate and transmit data to another computer, a null modem connection can be used to "simulate" the behavior of DCE equipment. This will be particularly important later on with some of the discussion in this series of articles, where you can experiment with writing some of your own serial communication software. In my own experience, I've had to write these "emulators" in many instances, either because the equipment that I was trying to communicate with wasn't finished, or it was difficult to obtain a sample of that equipment and all that I had available to me was the communication protocol specification.

2.2.3 Loopback Connectors

Sometimes instead of trying to communicate with another computer, you would like to be able to test the transmission equipment itself. One practical way of doing this is to add a "loopback" connector to the terminal device, like a PC with a serial data connection. This connector has no cable attached, but loops the transmit lines to the receive lines. By doing this, you can simulate both the transmission and receiving of data. Generally speaking, this is only done for actually testing the equipment, but can be used for testing software components as well. When this sort of connector is used, you will receive every byte that you transmit. If you separate out the transmission subroutines from the data capture subroutines, it can provide a controlled system for testing your application.

2.2.4 Protocol Analyzer

General

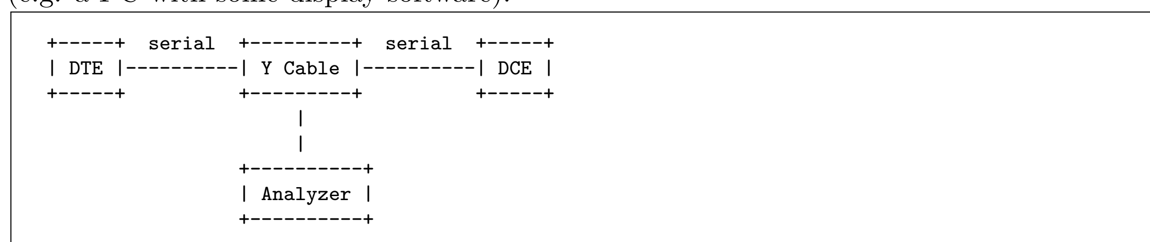
When it starts to get very difficult to examine the serial data being transmitted by the equipment, sometimes it is nice to be able to take a "snapshot" of the information being transmitted. This is done with a protocol analyzer of one kind or another.

What is done is a modification of the cabling that allows for a third computer to be able to simply read the data as it is being transmitted. Sometimes the communication protocol can get so complicated that you need to see the whole exchange, and it needs to be examined in "real-time" rather than going through some sort of software debugger. Another purpose of this is to examine the data exchange for purposes of doing some reverse engineering if you are trying to discover how a piece of equipment works. Often, despite written specifications, the actual implementation of what is occurring when transmitting data can be quite a bit different than what was originally planned. Basically, this is a powerful tool for development of serial communications protocols and software, and should not be ignored.

There are common ways to connect a protocol analyzer, which are discussed in the following.

Y "Cable"

A *Y "Cable"* is not just some cable, but also contains electronics - assuming it is not a low quality cable. It is supposed to be placed in between a serial line and it mirrors all signals on a third connector. This third connector can then be connected to a protocol analyzer (e.g. a PC with some display software):



It is recommended not to use a passive Y cable. Such a cable overloads the transmitters at the DTE and DCE, which might result in the **destruction of the transmitters**. The RS-233 standard requires that transmitters are short-circuit safe. However, modern, highly integrated equipment might no longer be compliant to that particular aspect of the standard.

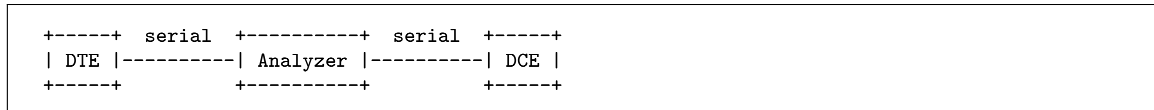
Often, the line going to the analyzer is also just a serial line, and the analyzer is a PC with a serial interface and some display software. The disadvantage of such a simple Y cable solutions is that it only supports half-duplex communication. That is, only one site (DTE or DCE) can talk at any time. The reason for this is that the two TX lines from the DTE and DCE are combined into one TX line going to the analyzer. If the DTE and the DCE both send at the same time, their signals get mixed up on the third line going to the analyzer, and the analyzer probably doesn't see any decodable signal at all.

See `http://www.mmvisual.de/fbintermdspy.htm` for an example of some simple circuitry for a Y cable.

More advanced Y cable solutions provide the TX data from the DTE and DCE separately to the analyzer. Such analyzers are capable of displaying full-duplex communication. Advanced professional systems not only display the decoded digital information, but also monitor the analog signal levels and timing.

Man-in-the-Middle

In this scenario the analyzer sits in the middle between the DTE and DCE. It is basically some device (e.g. a PC) with two serial interfaces. The analyzer mirrors each signal from one site to the other site, and also displays the traffic.



In principle, a simple version of such an analyzer can be built with any PC with two serial interfaces. All that is needed is some software, which is not too difficult to write. Such a device will, however, lack a convenient feature. Professional analyzers are able to auto-sense the speed of the serial communication. A home made solution needs to be configured to match the speed of the serial communication. Professional devices are also optimized to ensure minimal delay in the circuitry. Also, a simple homegrown, PC-based analyzer can't be used to analyze faults due to signal voltage level problems. Nevertheless, any kind of protocol analyzer is much better than nothing at all. Even the most simple analyzer is very useful.

Others

See Setting up a Development Environment (for modem development)[1] for some more information.

2.2.5 Breakout Box

An RS232 breakout box (a BOB) is a rather nifty piece of hardware which usually combines a number of functions into one. It basically consist of two RS232 connectors, and a patch field (or switches) which allows to change the wiring between the connectors. A patch field and small pieces of wires are preferable over (DIP) switches alone, since the patch field allows access to the signals for other purposes, too.

1 http://en.wikibooks.org/wiki/Serial_Programming%3AModems%20and%20AT%20Commands%23Setting%20up%20a%20Development%20Environment

A breakout box is very useful if the pinout (DTE/DCE) of a particular device is not known. The patch field allows to quickly change the wiring from a straight connection[2] to a null modem[3] connection, or to set up a loopback connection[4].

Since the patch field provides access to all signals it also allows to use the breakout box to connect a protocol analyzer[5]. Better breakout boxes also provide some signal level information on their own, by having LEDs who inform about the signal voltage. This information is useful when trying to identify an unknown pinout. High-end BOBs contain circuitry to measure ground potential difference and pulse traps circuitry to find signal glitches.

Commercial breakout boxes are available in many varieties. It is also possible to build a useful BOB from a handful of simple parts on a circuit board. The patch field can be made from DIL IC sockets, and the wiring of the LEDs is simple if 2-pin dual-color LEDs are used (3-pin LEDs will not work). Each signal line should be connected via such an LED and a 680 Ohm resistor in serial to GND (Signal Ground). The home-made breakout-box is completed with a couple of RS232 connectors, possibly also one to attach a protocol analyzer and some simple metal or plastic case.

2.2.6 Character Sequence Generator

Another nifty piece of hardware and/or software which is useful for developing and testing serial applications and equipment is a character sequence generator. Such a generator produces a repeated sequence of serial line data. For example such a generator might repeat the famous "The quick brown fox ..." sentence in an an endless loop. Another common test sequence is the generation of all 8-bit codes from 0x00 to 0xFF in a loop. Such a loop contains all 7-bit ASCII and 8-bit ISO Latin 1 characters, plus the first 32 non-printable control characters and can e.g. reveal decoding errors or transmission errors. Also very common is a modem test sequence, using generic modem commands (Serial Programming:Modems and AT Commands[6]) to build up a modem connection, send some data and tear the modem connection down in a loop.

Commercial hardware character generators provide a heap of additional features, often combined with a protocol analyzer. As such they are rather expensive. However, just like with a BOB, it is possible to build a useful DIY character sequence generator for small cash. This can either happen with software on a normal computer (some simple endless software loop sending the same data again and again to a serial interface), or with a few pieces of cheap electronic components. Some small stand-alone hardware is often more convenient in the field and in development for quick tests than e.g. a PC or laptop with some software.

2 http://en.wikibooks.org/wiki/Serial_Programming%3ARS-232_Connections%23Straight_Serial_Connections
3 http://en.wikibooks.org/wiki/Serial_Programming%3ARS-232_Connections%23Null_Modems
4 http://en.wikibooks.org/wiki/Serial_Programming%3ARS-232_Connections%23Loopback_Connectors
5 http://en.wikibooks.org/wiki/Serial_Programming%3ARS-232_Connections%23Protocol%20Analyser
6 http://en.wikibooks.org/wiki/Serial%20Programming%3AModems%20and%20AT%20Commands

A simple classic hardware character generator basically consists of a baud-rate generator, a UART (Serial Programming:8250 UART Programming[7]), an (E)EPROM, a binary counter and a line driver (Serial Programming:MAX232 Driver Receiver[8]). Typically, each of these components is a simple single IC. The (E)EPROM is supposed to contain the character sequence(s). The baud-rate generator drives the UART and the binary counter. The binary counter drives the address lines of the (E)EPROM. The result is that the character sequence is produced at the data lines of the (E)EPROM. These data lines are feed into the UARTs input. The UARTs output is connected to the serial line driver. All this can be easily fitted on a small prototype board in a simple case.

A more modern hardware character generator can be build around one of these small micro controllers (e.g. Atmel AVR[9]). This is particularly easy, since these micro controllers already contain serial interfaces, and just require a little bit of serial programming - which is the topic of this book[10].

2.3 Connection Types

If you wanted to do a general RS-232 connection, you could take a bunch of long wires and solder them directly to the electronic circuits of the equipment you are using, but this tends to make a big mess and often those solder connections tend to break and other problems can develop. To deal with these issues, and to make it easier to setup or take down equipment, some standard connectors have been developed that is commonly found on most equipment using the RS-232 standards.

These connectors come in two forms: A male and a female connector. The female connector has holes that allow the pins on the male end to be inserted into the connector.

2.3.1 EIA/TIA 574: "DB-9"

This is a female "DB-9" connector (properly known as DE9F):

7 http://en.wikibooks.org/wiki/Serial%20Programming%3A8250%20UART%20Programming
8 http://en.wikibooks.org/wiki/Serial%20Programming%3AMAX232%20Driver%20Receiver
9 http://en.wikibooks.org/wiki/Atmel%20AVR
10 http://en.wikibooks.org/wiki/Programming%3ASerial%20Data%20Communications

Figure 1 Female DB-9 Serial Connector

The female DB-9 connector is typically used as the "plug" that goes into a typical PC. If you see one of these on the back of your computer, it is likely not to be used for serial communication, but rather for things like early VGA or CGA monitors (not SVGA) or for some special control/joystick equipment.

And this is a male "DB-9" connector (properly known as DE9M):

Figure 2 Male DB-9 Serial Connector

This is the connector that you are more likely to see for serial communications on a "generic" PC. Often you will see two of them side by side (for COM1 and COM2). Special equipment that you might communicate with would have either connector, or even one of the DB-25 connectors listed below.

2.3.2 RS-232C: DB-25

This is a female DB-25 connector (also known as DB25F):

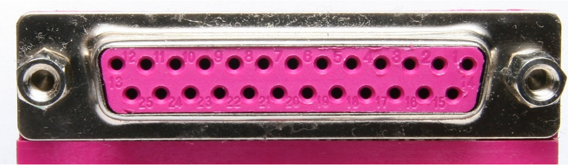


Figure 3 Female DB-25 Serial Connector

This DB25S is what you normally find on an IBM compatible PC used as the parallel (printer) port. It is also on the computer end of a modem cable in older PCs that have 25 pin serial port connectors. This connector type is also used frequently for equipment that conforms to RS-232 serial data communication as well, so don't always assume if you see one of these connectors that it is always parallel. When the original RS-232 specification was written, this was the kind of connector that was intended, but because many of the pins were seldom if ever used, IBM PC compatible serial ports were later switched to the DB-9 DE9S connectors carrying all the required signals as on the DB connectors in the original IBM-PC. (Yes, this is comparatively recent equipment for this standard).

This is a male DB-25 connector (also known as DB25M):

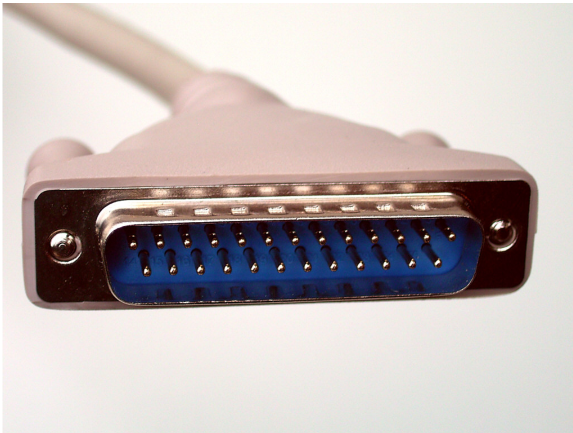


Figure 4 Male DB-25 Serial Connector

Male DB-25 connectors are usually used on one end of a PC printer cable for parallel data communication, which is beyond the scope of this series of articles. The DB25P is also used on the the modem end of an external modem cable. You should be aware that this connector is also used for serial communications on many different types of equipment, using many different types of communications protocols. In fact, if you have a random piece of equipment that you are trying to see how it works, you can presume that it is a piece of serial equipment. Hacking random connectors is also beyond the scope of this document, but it can be an interesting hobby by itself.

2.3.3 mini-stereo plug connector

This is a male mini-stereo plug connector:

Figure 5 mini-stereo_plug connector

Some digital cameras and calculators come with a cable that has a mini-stereo plug connector on the end the plugs into the camera, and a DB-9 connector on the end that plugs into the PC.

It is a poor connector, as it short circuits segments while being plugged/unplugged.

The "PicAXE" systems use `http://profmason.com/?p=218`

- 1: base ring: ground (pin 5 of DB9)
- 2: middle ring: serial output from PicAXE to serial input of PC (pin 2 of DB9)
- 3: tip of pin: serial output of PC to serial input of PicAXE (pin 3 of DB9)

2.3.4 RS-232D: RS232 on RJ45

RS-232D defines a standard connector much smaller than a DB-9 plug. `http://zytrax.com/tech/layer_1/cables/tech_rs232.htm#rj45`.

(RS-232 on a RJ45 modular jack is also known as "EIA/TIA - 561")

2.3.5 RS232 on RJ11

Is there a standard for connecting the TX, RX, GND of RS-232 to the 4 pins of a RJ11 connector ?

- Luhan Monat[11] uses DB9-5 ---> RJ11-1; DB9-3 ---> RJ11-2; DB9-2 ---> RJ11-3. (RJ11-2 and RJ11-3 are the "inner pair").
- Paul Campbell[12] says "I wired the GND to the yellow line, TXD to the black line and RXD to the red line."

11 `http://mondo-technology.com/upp.html`
12 `http://www.taniwha.com/~paul/fc/ass2.0.html`

2.4 Wiring Pins Explained

The wiring of RS-232 devices involves first identifying the actual pins that are being used.

Please note also that in the "PC COMx Port context" end of things some signals are 'inputs' while others are 'outputs' while in the "Modem context" those same signal names referred to now become as 'outputs' where they were just before 'inputs' and vice versa. That is where much confusion has arisen from over the years, as the 'Input' or 'Output' -sense- nature is not noted in most diagrams on the subject in general, yet in the real world two 'Out' pins seldom can ever work in harmony in RS-232 related +-[3-10]V stuff where the range from -3V to +3V is not a true high or low, except to possibly burden drivers towards their undesired burnout.

Here is how a **female** DB-9 connector is numbered (Note, the connector on a computer is usually a **male** connector, so it is mirrored compared to the following image):

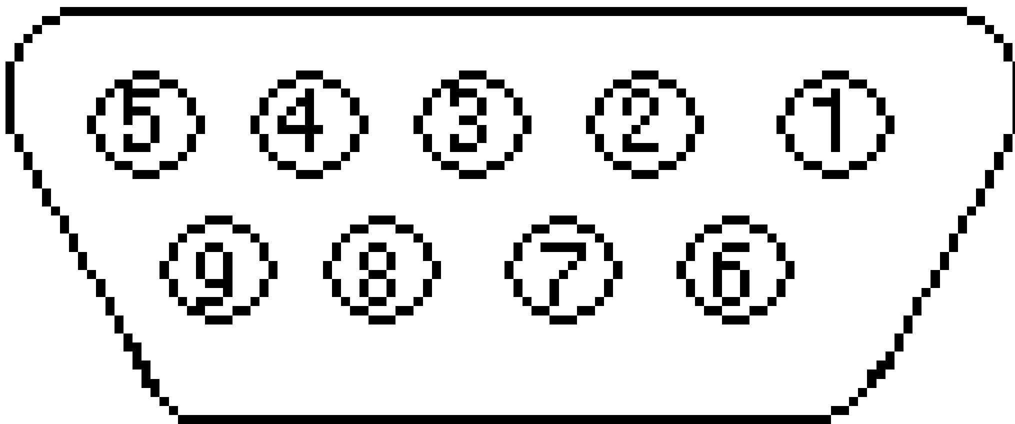


Figure 6 DB-9 Female Pinout Diagram

If the numbers are hard to read, it starts at the top-right corner as "1", and goes left until the end of the row and then starts again as pin 6 on the next row until you get to pin 9 on the bottom-left pin. "Top" is defined as the row with 5 pins.

Here are what each pin is usually defined as on the PC COMx end of things:

9-pin	**25-pin**	**pin definition**	Direction (PC view)
1	8	DCD (Data Carrier Detect)	input
2	3	RX (Receive Data)	input
3	2	TX (Transmit Data)	output
4	20	DTR (Data Terminal Ready)	output
5	7	GND (Signal Ground)	-
6	6	DSR (Data Set Ready)	input
7	4	RTS (Request To Send)	output
8	5	CTS (Clear To Send))	input
9	22	RI (Ring Indicator)	input

One thing to keep in mind when discussing these pins and their meaning, is that they are very closely tied together with modems and modem protocols.

Whenever interconnecting any serial ports it will be well to note that whatever the case, it should always follow that only one <output> should ever be tied to one or more <inputs> generally speaking. Further, be it noted that signal names at the COMx end will generally be opposite of the <in>-<out> -sense- at the modem end of things, even though carrying the same mnemonic names.

Often you don't have a modem attached in the loop, but you still treat the equipment as if it were a modem on a theoretical level. At least such that you minimally have an <output> going to every in some manner, with no two <outputs> in conflict or without any 'floating' <inputs> tied to no <output> at all.

The following are more formal explanations regarding each signal function in the general sense of its use:

2.4.1 DCD (Data Carrier Detect)

This is a signal to indicate from the communications equipment (DCE) that the phone line is still "connected" and receiving a carrier signal from the modem at the other end. Presumably well-written software or serial equipment could detect from this logic state when the telephone has been "hung up" on the other end. Null-modems often tie DCD to DTR at each end since there is no carrier signal involved.

2.4.2 RX (Receive Data)

Input to receive the data.

2.4.3 TX (Transmit Data)

The reverse of RX, this is where the terminal equipment (DTE) is transmitting serial data, using the same format and protocol that the receiver is expecting. More on the exact protocol further below. Like RX, think along the lines of "Terminal Transmit" when designing equipment that will be using this pin.

2.4.4 DTR (Data Terminal Ready)

Basically a signal from the DTE that says "Hello!, I'm ready if you are". This is a general indicator to the DCE that the terminal is ready to start sending and receiving data. If there is some initialization that needs to happen in the communications equipment, this is a way for the terminal equipment to "boot" the receiving equipment. In an null modem setup this signal is often connected to DCD, so the device signals itself that an (imaginary) carrier has been detected, indication that the transmission line is up.

2.4.5 GND (Signal Ground)

This is an interesting pin to look at. What it does is try to make a common "ground" reference between the equipment that is being connected to compare the voltages for the other signals. Normally this is a good thing, because sometimes different pieces of equipment have different power supplies and are some distance away. The not so pleasant thing about this wire is that it usually is a physical piece of copper that can conduct electricity that is not normally supposed to go down the wire, like a short-circuit or worse yet a bolt of lightning (it happens far more often that you would normally think for this sort of equipment). That can fry both the DCE as well as the DTE. Things like fiber converters and ground isolators can help prevent this from happening, but can still be something to worry about. Over short distances this is generally not a problem.

2.4.6 DSR (Data Set Ready)

This is the counterpart to DTR with the communications equipment (or computer peripheral on the serial line). When the DTR is sent as a signal, the communications equipment should change this signal to logic "1" to indicate that it is ready to communicate as well. If the DCE goes through a "boot" sequence when the DTR gets signaled, it should not signal DSR until it is complete. But many connectors "hard wire" this pin to be directly connected to the DTR pin at each end to reduce the number of wires needed in the cable. This can be useful for connecting devices using existing telephone wires, but prevents applications from using the DTR and DSR for handshaking.

2.4.7 RTS (Request To Send)

Setting the RTS signal to logic "1"[13] indicates to the DCE that the DTE wants to send it data. Resetting the RTS signal to logic "0"[14] indicates to the DCE that the DTE has no more data to send.

2.4.8 CTS (Clear To Send)

This is the response signal from the DCE regarding if the terminal equipment should be transmitting any data. When this signal is at logical "1"[15], the terminal is "permitted" to transmit data. Like the DTR/DSR pins, this one can be directly connected to the RTS pin to reduce the number of wires needed, but this eliminates the possibility of hardware flow control. Some software ignores this pin and the RTS pin, so other flow control systems are also used. That will be explained when we get to actual software.

13 `http://en.wikipedia.org/wiki/RS-232C%23Voltage_levels`
14 `http://en.wikipedia.org/wiki/RS-232C%23Voltage_levels`
15 `http://en.wikipedia.org/wiki/RS-232C%23Voltage_levels`

2.4.9 RI (Ring Indicator)

Again, thinking back to a telephone modem, this is a signal that indicates that the telephone is "ringing". Generally, even on a real telephone modem, this is only occasionally set to -15V for the signal. Basically, when you would normally be hearing a "ring" on your telephone, this pin would be signaled. On Null-modems, often this wire isn't even connected to anything. If you really are connected to a real modem, this does have some strong uses, although there are other ways to have the terminal equipment (like a PC connected to an external modem) be informed that there are ways to communicate this information through the data pins as well. This will be covered lightly in the software section.

2.4.10 Other RS-232 Pins

There are other pins that the DB-25 has implemented that the DB-9 doesn't normally use, such as a secondary transmit and receive pin, Secondary CTS/RTS for those alternate pins, a -15V signal for power, a clock, and a couple of other good ideas as well. The problem with implementing all of these pins is that you also need to run separate wires, and a full set of DB-25 connectors would also mean having 25 physical wires going the full distance between the DTE and DCE. If this is more than a foot or so, it gets to be a big hassle, particularly if you are going through walls or in a more permanent setting. If the wrong wire gets clipped in the bundle, the whole thing must be restrung again, or you must go through wire testing like the old-fashioned telephone linemen used to have to do when fixing a phone distribution box. Often only three physical copper lines are used to connect the DTE to DCE, and that is simply RX, TX, and GND. The rest can be easily "faked" on the connector end in a manner sufficient for most software and hardware applications.

2.5 Baud Rates Explained

Baud and BPS (Bits Per Second) are usually not the same thing, although they are often used interchangeably, particularly in marketing literature. There are several ways to determine what the actual data rate of a particular piece of equipment is, but in popular marketing literature, or even general reference texts, they will almost always refer to "Baud Rate", even if they are referring to bits per second.

Baud means the number of changes to the transmission media per second in a modulated signal. If each transmission event contains more than one bit of information, then Baud and BPS are not the same. E.g. if each event contains two bits (two bits modulated in an event), then the BPS of such a transmission would be twice as large as the Baud rate. This is not a theoretical case. Typical "high speed" modems use sophisticated modulation on the telephone line, where the bit rate and Baud rate differ significantly on the line. It is important to know this when you build measurement equipment, decoders (demodulators), encoders (modulators), and all sorts of transmission equipment for a particular protocol.

However, software developers typically like to ignore the difference of bit rate and baud rate, because a bit can either have the value true or false - an "event" (a bit) always only has two possible states. They have no basic unit which can e.g. hold four different states. In other words, on the software site the modulation has already been flattened by the

demodulator. If a modulation was used which can e.g. transmit 8 bits in an event, the software developer sees them already as a series of 8 consecutive bits, each either true or false. The demodulator took care of that. When it got an event it turned the single 8-bit event into eight single-bit events. Software developers don't see the original single entity with 256 different states (voltages, phases). Since the modulation has been flattened they don't experience the difference between Baud rate and bit rate any more. This is not the fault of the people who defined a Baud or a BPS. It is just a (welcome) limitation of digital computer hardware.

Baud is actually a shortened term named in honor of Émile Baudot, a French inventor of early teleprinter machines that replaced the telegraph key using Morse Code. Basically two typewriters that could be connected to each other with some wires. He came up with some of the first digital character encoding schemes, and the character codes were transmitted with a serial data connection. Keep in mind this was being done largely before computers were invented. Indeed, some of these early teleprinter devices were connected to the very first computers like the ENIAC or UNIVAC, simply because they were relatively cheap and mass produced at that point.

In order for serial data communication to happen, you need to agree on a clock signal, or baud rate, in order to get everything to be both transmitted and received properly. This is where the language purists get into it, because it is this clock signal that actually drives the "baud rate". Let's start more at the beginning with Émile Baudot's teleprinters to explain baud rate.

Émile's early teleprinters used 5 data bits and 1 stop bit to transmit a character. We will go onto formatting issues in a second, but what is important is that six signals are sent through a wire in some fashion that would indicate that a character is transmitted. Typically the equipment was designed to run at 50 baud, or in other words the equipment would transmit or receive a "bit" of data 50 times per second. Not coincidentally, French power systems also ran on an alternating current system of 50 Hz, so this was an easy thing to grab to determine when a new character should be transmitted.

Teleprinters evolved, and eventually you have Western Union sending teleprinter "cablegrams" all around the world. If you hear of a TELEX number, this is the relic of this system, which is still in use at the present time, even with the Internet. By rapidly glossing over a whole bunch of interesting history, you end up with the United States Department of Justice (DOJ) in a lawsuit with AT&T. Mind you this was an earlier anti-trust lawsuit prior to the famous/infamous 1982 settlement. The reason this is important is because the DOJ insisted that Western Union got all of the digital business (cable grams... and unfortunately this got to be read as computer equipment as well), and AT&T got modulated frequencies, or in other words, you could talk to your mother on Mother's Day on their equipment. When computers were being built in the 1950s, people wanted some way to connect different pieces of computer equipment together to "talk" to each other. This finally resulted in the RS-232 standard that we are discussing on this page.

While Western Union was permitted to carry digital traffic, often the connections weren't in or near computer centers. At this time AT&T found a loophole in the anti-trust settlement that could help get them into the business of being a "carrier" of computer data. They were also offering to transmit computer data at rates considerably cheaper than Western Union was going to charge. Hence, the modem was born.

2.5.1 Modems Explained

The long description of a modem is a "Modulator/Demodulator", and this description is important. Since AT&T could only carry "tones", like music from a radio network or the voice of your mother, they created a device that would electronically create "music" or "tones" that could be carried on their network. They would then take a computer "1" or "0" and "modulate" the bit to a frequency, like say 2600 Hz. (The exact tones varied based on baud rate and other factors, but there were exact frequency specs here.) A matching device would be able to look for that "note" or "tone" in the "music" and be able to convert that back to a computer "1" or "0", or in other words, demodulate the music. Since all you and your buddy on each end of the telephone are only playing music to each other, it was legal for AT&T to have that music on their network. That only computers could possibly understand this music is besides the point, and the DOJ turned a blind eye on the whole practice, despite objections from Western Union.

The original modems you could rent were AT&T Bell 103 modems. These were clunky boxes about the size of a shoe box that had a bunch of switches on the outside and an RS-232 cable that connected to the computer equipment you were using. These boxes were designed for the old-fashioned handset telephones and had pieces of rubber that would go around the "speaker" and "mic" portion of the telephone (no direct copper connection to the telephone equipment back then). If you wanted to dial the telephone, you had to use the rotary dial on the phone itself... the computer didn't have access to that sort of equipment. Keep in mind that the FCC regulated just about everything that happened with phone equipment, and AT&T owned everything related to telephones. You even had to "rent" the modem from AT&T, and that rental charge was on your monthly phone bill.

The Bell 103 was originally 110 baud, although it eventually had a switch to "move up" to 220 baud. 300 baud modems were also fairly common throughout the 1960's and 1970's. Keep in mind that AT&T (or your local phone company) was the only company you could even rent a modem from, whether you wanted one or not. By 1982, modems were so commonly used and the POTS telephone network so widespread that this same system of sending "music" over the telephone has been preserved, even though the legal reasons for doing it are no longer valid. With the advent of ISDN and DSL lines, this is no longer the case and the phone companies are now sending pure digital signals instead. This is also why DSL lines can carry much more data than an ordinary phone line, even though it is the same pair of copper wires going into your home.

When modems started going to very high speeds, they hit a brick wall of sorts. It was decided back in the 1950's that telephone equipment would only have to carry tone signals going to about 10kHz. For normal voice conversations this is sufficient, and you can even tell the difference between a man and a woman on the telephone. The problem comes in that this means the highest normal "baud rate" that you can send over a home telephone network is about 9600 baud, usually about 4800 baud, because the telephone equipment itself is going to be dropping "bits" as you switch from one tone to another. Without going into the heavy math, you need to have at least one full "sound wave" in order to be able to distinguish one tone or note from another. Modem manufacturers did think of something else that could be done to overcome this limitation, however. Instead of just sending one tone at a time, you could play a whole "chord", or several distinct tones at the same time. Finally back to baud vs. bits per second. With higher speeds, instead of simply sending

only one bit, you are sending two or as many as sixteen bits at the same time with varying "chords" of "music". This is how you get a 56K BPS modem, even though it is still only transmitting at 9600 baud.

More about modems in Serial Programming:Modems and AT Commands[16].

2.6 Signal Bits

There are four sets of transmission bits that are used in the RS-232 standard. The positioning of these bits in the RS-232 data stream is all that distinguishes one bit from the other. This is also where serial communication really hits the "metal", because each bit follows in a sequence, or in a serial fashion. All of the other wires, pins, baud rate, and everything else is to make sure that these bits can be understood. Keep in mind that at this point the entire protocol is based on the transmission of a single character. Multiple characters can be sent, but they are a sequence of single character transmission events. How the characters relate is based on what the software does with the data on the next protocol "layer".

2.6.1 Start Bit

When a transmission line is not sending anything, it remains in a logical state of "1", or -15V on the wire. When you want to send a character, you start by changing the voltage to +15V, indicating a logical "0" state. Each subsequent bit is based on the baud rate that is established for communication between each device. This bit signals that the receiving device should start scanning for subsequent bits to form the character.

2.6.2 Data Bits

This is the primary purpose of serial communications, where the data actually gets sent. The number of bits here can vary quite a bit, although in current practice the number of bits typically transmitted is eight bits. Originally this was five bits, which was all that the early teleprinters really used to make the letters of the Alphabet and a few special characters. This has implications for Internet protocols as well, because early e-mail systems transmitted with only seven bits when they were connected over some RS-232 links. This worked because the early character encoding schemes, mainly ASCII, only used seven bits to encode all characters commonly used for the English language. Because computer components work best on powers of 2 (2,4,8,16,32, etc.), eight bits became more commonly used for data storage of individual characters. Unicode and other coding schemes have moved this concept forward for languages other than English, but eight bits still is a very common unit for transmitting data, and the most common setting for RS-232 devices today.

The least significant bit (LSB) is transmitted first in this sequence of bits to form a character.

16 http://en.wikibooks.org/wiki/Serial%20Programming%3AModems%20and%20AT%20Commands

2.6.3 Parity Bit

To help perform a limited error check on the characters being transmitted, the parity bit has been introduced. Parity can detect some transmission errors but not correct. The value of the parity bit depends on the number of bits set to "1" in the string of data bits.

There are four different kinds of parity configuration to consider:

Odd Parity

When the sum of bits ends up coming up with an odd number (like the sequence 01110110), this bit will be set to a logical state of "1".

Even Parity

This uses the formula of trying to determine if there are an even number of bits set to "1". In this regard, it is the exact opposite state of the Odd Parity. For e.g., for a frame with seven bits that has an odd number of ones, the parity bit will be set to one. So essentially, the entire byte, including parity must have an even number of ones for even parity.

Mark Parity

Using this concept, the transmission protocol is essentially ignoring the parity bit entirely. Instead, the transmission configuration is sending a logical "1" at the point that a parity bit should be sent, regardless of if the sequence should have an odd or even count. This configuration mode is useful for equipment that may want to be testing parity checking software or firmware in the receiving equipment.

Space Parity

The opposite of Mark parity, this sends a logical "0" for the parity checksum. Again, very useful for equipment diagnostics.

Parity None

This isn't really a parity formula, but rather an acknowledgment that parity really doesn't work, so the equipment doesn't even check for it. This means the parity bit isn't even used. This can cause, in some circumstances, a slight increase in the total data throughput. More on that below.

2.6.4 Stop Bits

This really isn't a bit at all, but an agreement that once the character is sent that the transmitting equipment will return to a logical "1" state. The RS-232 specification requires

this logical state of "1" to remain for at least one whole clock cycle, indicating that the character transmission is complete. Sometimes the protocol will specify two stop bits. One reason that this might be done is because the clock frequencies being used by the equipment might have slightly different timing, and over the course of hundreds or thousands of characters being transmitted the difference between two clocks on the two different pieces of equipment will cause the expected bits to be shifted slightly, causing errors. By having two stop bits the transmission is slightly slower, but the clock signals between the two pieces of equipment can be coordinated better. Equipment expecting one stop bit can accept data transmitted by equipment sending two stop bits. It won't work the other way around, however. This is something to try if you are having problems trying to get two pieces of equipment to communicate at a given baud rate, to add the second stop bit to the transmitter.

2.6.5 Data Transmission Rates

We got into a discussion of baud rate vs. bits per second. Here is where baud as the number of bits being transmitted is still off, even if the nominal bits per second is also the same as the baud rate. By adding start bits, stop bits, and parity bits, that is going to add overhead to the transmission protocol. All digital transmission protocols have some sort of overhead on them, so this shouldn't be that much of a surprise. As we get more into data packets and other issues, the actual amount of data being transmitted will drop even further.

Keep in mind that if you are transmitting with 6 data bits, 2 Stop bits, and Even Parity, you are transmitting only six bits of data and four other bits of extra information. That means even with 9600 baud, you are only transmitting 5,760 bits of data per second. This really is a big difference, and that is still only raw bits once it gets through the actual serial communications channel. A more typical 8 data bits, 1 Stop Bit, No Parity will be a little bit better at 9600 baud, with eight bits of data and only two bits used for overhead. That gives a total throughput of 7,680 bits per second. A little bit better, but you can't simply presume that the baud rate indicates how much data is going to be transmitted.

2.7 Relationship of Baud Rate to Maximum Distance

There are physical limits to how far serial data communication can occur over a piece of wire. When you apply a voltage onto a wire it takes time for that voltage to traverse the wire, and there are other unstable conditions that happen when you send a "pulse" down the wire and change voltages too quickly. This problem is worse as wires become longer and the frequency (i.e. baud rate) increases. This distance can vary based on a number of factors, including the thickness of the wires involved, RF interference on the wires, quality of the wires during the manufacturing process, how well they were installed... e.g., are there any "kinks" in the wires that force it into a sharp bend, and finally the baud rate that you are transmitting the data.

This table presumes a fairly straight and uniform cable that is typical for most low-voltage applications (i.e., not a power circuit that uses 110V to run your refrigerator, toaster, and television). Typically something like a CAT-5 cable (also used for local networks or phone lines) should be more than sufficient for this purpose.

Baud Rate	Maximum Distance (in feet)	Maximum Distance (in meters)
2400	3000	914.4
4800	1000	304.8
9600	500	152.4
19200	50	15.24

The distance limitation can be mitigated. There are "short haul modems" that can extend this distance to several miles of cable. There are also telephone lines, or conventional modems, and other long-distance communications techniques. There are other ways to handle data in situations like this, and those signals can be converted to simple RS-232 data formats that a typical home computer can interpret. Distance still can be a limiting factor for communication, although when you are talking about distances like to Saturn for the Cassini mission, serial data communication has other issues involved than just data loss due to cable length. And yes, NASA/ESA is using serial data communication for transmitting those stunning images back to Earth.

2.8 External References

w:Serial cable[17]

- RS-232 wiring standards explained[18]
- RS-232 connection types explained[19]
- Wikipedia article on RS-232[20]
- RS-232 standards explained by HW-Server[21]
- Serial Pinouts (D25 and D9 Connectors)[22] (also has more technical information about the UARTs used in PCs)
- RS232 Connections, and wiring up serial device[23] has several diagrams, including one showing how to let one PC monitor the serial communication between 2 other RS232 devices.
- Lammert Bies, RS232 Specifications and standard[24] Includes technical specs on RS-232 signals and more detailed information about parity checking.
- Tronisoft's Printable ASCII Serial Port Crib Sheets[25]
- jSSC library (Java Simple Serial Connector). Work under Win32 and Win64[26]

17 http://en.wikipedia.org/wiki/Serial%20cable
18 http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html
19 http://www.arcelect.com/rs232.htm
20 http://en.wikipedia.org/wiki/RS-232C
21 http://hw-server.com/rs232-overview-rs232-standard
22 http://www.beyondlogic.org/serial/serial.htm#2
23 http://airborn.com.au/serial/rs232.html
24 http://www.lammertbies.nl/comm/info/RS-232_specs.html
25 http://www.tronisoft.com/rs232info/ASCII_serial_port_crib_sheets.pdf
26 http://code.google.com/p/java-simple-serial-connector/

2.9 Other Serial Programming Articles

Typical RS232-Hardware Configuration[27]

Category:Serial Programming[28]

27 http://en.wikibooks.org/wiki/Serial%20Programming%3ATypical%20RS232-Hardware%20Configuration
28 http://en.wikibooks.org/wiki/Category%3ASerial%20Programming

3 8250 UART Programming

3.1 Introduction

Finally we are moving away from wires and voltages and hard-core electrical engineering applications, although we still need to know quite a bit regarding computer chip architectures at this level. While the primary focus of this section will concentrate on the 8250 UART, there are really three computer chips that we will be working with here:

- 8250 UART
- 8259 PIC (Programmable Interrupt Controller)
- 8086 CPU (Central Processing Unit)

Keep in mind that these are chip families, not simply the chip part number itself. Computer designs have evolved quite a bit over the years, and often all three chips are put onto the same piece of silicon because they are tied together so much, and to reduce overall costs of the equipment. So when I say 8086, I also mean the successor chips including the 80286, 80386, Pentium, and compatible chips made by manufacturers other than Intel. There are some subtle differences and things you need to worry about for serial data communication between the different chips other than the 8086, but in many cases you could in theory write software for the original IBM PC doing serial communication and it should run just fine on a modern computer you just bought that is running the latest version of Linux or Windows XP.

Modern operating systems handle most of the details that we will be covering here through low-level drivers, so this should be more of a quick understanding for how this works rather than something you might implement yourself, unless you are writing your own operating system. For people who are designing small embedded computer devices, it does become quite a bit more important to understand the 8250 at this level.

Just like the 8086, the 8250 has evolved quite a bit as well, e.g. into the 16550 UART. Further down I will go into how to detect many of the different UART chips on PCs, and some quirks or changes that affect each one. The differences really aren't as significant as the changes to CPU architecture, and the primary reason for updating the UART chip was to make it work with the considerably faster CPUs that are around right now. The 8250 itself simply can't keep up with a Pentium chip.

Remember as well that this is trying to build a foundation for serial programming on the software side. While this can be useful for hardware design as well, quite a bit will be missing from the descriptions here to implement a full system.

3.2 8086 I/O ports

We should go back even further than the Intel 8086, to the original Intel CPU, the 4004, and its successor, the 8008. All computer instructions, or op-codes, for the 8008 still function in today's Intel chips, so even port I/O tutorials written 30 years ago are valid today. The newer CPUs have enhanced instructions for dealing with more data more efficiently, but the original instructions are still there.

When the 8008 was released, Intel tried to devise a method for the CPU to communicate with external devices. They chose a method called I/O port architecture, meaning that the chip has a special set of pins dedicated to communicating with external devices. In the 8008, this meant that there were a total of sixteen (16) pins dedicated to communicating with the chip. The exact details varied based on chip design and other factors too detailed for the current discussion, but the general theory is fairly straightforward.

Eight of the pins represent an I/O code that signaled a specific device. This is known as the I/O port. Since this is just a binary code, it represents the potential to hook up 256 different devices to the CPU. It gets a little more complicated than that, but still you can think of it from software like a small-town post-office that has a bank of 256 PO boxes for its customers.

The next set of pins represent the actual data being exchanged. You can think of this as the postcards being put into or removed from the PO boxes.

All the external device has to do is look for its I/O code, and then when it matches what it is "assigned" to look for, it has control over the corresponding port. An pin signals whether the data is being sent to or from the CPU. For those familiar with setting up early PCs, this is also where I/O conflicts happen: when two or more devices try to access the same I/O port at the same time. This was a source of heartburn on those early systems, particularly when adding new equipment.

Incidentally, this is very similar to how conventional RAM works, and some CPU designs mimic this whole process straight in RAM, reserving a block of memory for I/O control. This has some problems, including the fact that it chews up a portion of potential memory that could be used for software instead. It ends up that with the IBM PC and later PC systems, both I/O methods are used extensively, so it really gets complicated. For serial communication, however, we are going to stick with the port I/O method, as that is how the 8250 chip works.

3.2.1 Software I/O access

When you get down to actually using this in your software, the assembly language instruction to send or receive data to port 9 looks something like this:

```
out 9, ah ; sending data from register ah out to port 9
in ah, 9 ; getting data from port 9 and putting it in register ah
```

When programming in higher level languages, it gets a bit simpler. A typical C language Port I/O library is usually written like this:

```
char test;

test = 255;
outp(9,test);
inp(9,*test);
```

For many versions of Pascal, it treats the I/O ports like a massive array that you can access, that is simply named Port:

```
procedure PortIO(var Test: Byte);
begin
  Port[9] := Test;
  Test := Port[9];
end;
```

Warning!! And this really is a warning. By randomly accessing I/O ports in your computer without really knowing what it is connected to can really mess up your computer. At the minimum, it will crash the operating system and cause the computer to not work. Writing to some I/O ports can permanently change the internal configuration of your computer, making a trip to the repair shop necessary just to undo the damage you've done through software. Worse yet, in some cases it can cause actual damage to the computer. This means that some chips inside the computer will no longer work and those components would have to be replaced in order for the computer to work again. Damaged chips are an indication of lousy engineering on the part of the computer, but unfortunately it does happen and you should be aware of it.

Don't be afraid to use the I/O ports, just make sure you know what you are writing to, and you know what equipment is "mapped" to for each I/O port if you intend to use a particular I/O port. We will get into more of the specifics for how to identify the I/O ports for serial communication in a bit. Finally we are starting to write a little bit of software, and there is more to come.

3.2.2 x86 port I/O extensions

There are a few differences between the 8008 CPU and the 8086. The most notable that affects software development is that instead of just 256 port I/O addresses, the 8086 can access 65536 different I/O ports. In addition, besides simply sending a single character in or out, the 8086 will let you send and receive 16 bits at once. The 386 chips will even let you send and receive 32-bits simultaneously. The need for more than 65536 different I/O ports has never been a serious problem, and if a device needed a larger piece of memory, the Direct Memory Access (DMA) methods are available. This is where the device writes and reads the RAM of the computer directly instead of going through the CPU. We will not cover that topic here.

Also, while the 8086 CPU was able to address 65536 different I/O ports, in actual practice it didn't. The chip designers at Intel got cheap and only had address lines for 10 bits, which has implications for software designers having to work with legacy systems. This also meant that I/O port address $1E8 and $19E8 (and others... this is just an example) would resolve to the same I/O port for those early PCs. The Pentium CPUs don't have this limitation,

but software written for some of that early hardware sometimes wrote to I/O port addresses that were "aliased" because those upper bits were ignored. There are other legacy issues that show up, but fortunately for the 8250 chip and serial communications in general this isn't a concern, unless you happen to have a serial driver that "took advantage" of this aliasing situation. This issue would generally only show up when you are using more than the typical 2 or 4 serial COM ports on a PC.

3.3 x86 Processor Interrupts

The 8086 CPU and compatible chips have what is known as an interrupt line. This is literally a wire to the rest of the computer that can be turned on to let the CPU know that it is time to stop whatever it is doing and pay attention to some I/O situations.

Within the 8086, there are two kinds of interrupts: Hardware interrupts and Software interrupts. There are some interesting quirks that are different from each kind, but from a software perspective they are essentially the same thing. The 8086 CPU allows for 256 interrupts, but the number available for equipment to perform a Hardware interrupt is considerably restricted.

3.3.1 IRQs Explained

Hardware interrupts are numbered IRQ 0 through IRQ 15. IRQ means Interrupt ReQuest. There are a total of fifteen different hardware interrupts. Before you think I don't know how to count or do math, we need to do a little bit of a history lesson here, which we will finish when we move on to the 8259 chip. When the original IBM-PC was built, it only had eight IRQs, labeled IRQ 0 through IRQ 7. At the time it was felt that was sufficient for almost everything that would ever be put on a PC, but very soon it became apparent it wasn't nearly enough for everything that was being added. When the IBM-PC/AT was made (the first one with the 80286 CPU, and a number of enhancements that are commonly found on PCs today), it was decided that instead of a single 8259 chip, they would use two of these same chips, and "chain" them to one another in order to expand the number of interrupts from 8 to 15. One IRQ had to be sacrificed in order to accomplish this task, and that was IRQ 2.

The point here is that if a device wants to notify the CPU that it has some data ready for the CPU, it sends a signal that it wants to stop whatever software is currently running on the computer and instead run a special "little" program called an interrupt handler. Once the interrupt handler is finished, the computer can go back to whatever it was doing before. If the interrupt handler is fast enough, you wouldn't even notice that the handler has even been used.

In fact, if you are reading this text on a PC, in the time that it takes for you to read this sentence several interrupt handlers have already been used by your computer. Every time that you use a keyboard or a mouse, or receive some data over the Internet, an interrupt handler has been used at some point in your computer to retrieve that information.

3.3.2 Interrupt handlers

We will be getting into specific details of interrupt handlers in a little bit, but now I want to explain just what they are. Interrupt handlers are a method of showing the CPU exactly what piece of software should be running when the interrupt is triggered.

The 8086 CPU has a portion of RAM that has been established that "points" to where the interrupt software is located elsewhere in RAM. The advantage of going this route is that the CPU only has to do a simple look-up to find just where the software is, and then transfers software execution to that point in RAM. This also allows you as a programmer to change where the CPU is "pointing" to in RAM, and instead of going to something in the operating system, you can customize the interrupt handler and put something else there yourself.

How this is best done depends largely on your operating system. For a simple operating system like MS-DOS, it actually encourages you to directly write these interrupt handlers, particularly when you are working with external peripherals. Other operating systems like Linux or MS-Windows use the approach of having a "driver" that hooks into these interrupt handlers or service routines, and then the application software deals with the drivers rather than dealing directly with the equipment. How a program actually does this is very dependent on the specific operating system you would be using. If you are instead trying to write your own operating system, you would have to write these interrupt handlers directly, and establish the protocol on how you access these handlers to send and retrieve data.

3.3.3 Software interrupts

Before we move on, I want to hit very briefly on software interrupts. Software interrupts are invoked with the 8086 assembly instruction "int", as in:

int $21

From the perspective of a software application, this is really just another way to call a subroutine, but with a twist. The "software" that is running in the interrupt handler doesn't have to be from the same application, or even made from the same compiler. Indeed, often these subroutines are written directly in assembly language. In the above example, this interrupt actually calls a "DOS" subroutine that will allow you to perform some sort of I/O access that is directly related to DOS. Depending on the values of the registers, usually the AX register in the 8086 in this case, it can determine just what information you want to get from DOS, such as the current time, date, disk size, and just about everything that normally you would associate with DOS. Compilers often hide these details, because setting up these interrupt routines can be a little tricky.

Now to really make a mess of things. "Hardware interrupts" can also be called from "software interrupts", and indeed this is a reasonable way to make sure you have written your software correctly. The difference here is that software interrupts will only be invoked, or have their portion of software code running in the CPU, if it has been explicitly called through this assembly opcode.

3.4 8259 PIC (Programmable Interrupt Controller)

The 8259 chip is the "heart" of the whole process of doing hardware interrupts. External devices are directly connected to this chip, or in the case of the PC-AT compatibles (most likely what you are most familiar with for a modern PC) it will have two of these devices that are connected together. Literally fifteen wires come into this pair of chips, each wire labeled IRQ-0 through IRQ-15.

The purpose of these chips is to help "prioritize" the interrupt signals and organize them in some orderly fashion. There is no way to predict when a certain device is going to "request" an interrupt, so often multiple devices can be competing for attention from the CPU.

Generally speaking, the lower numbered IRQ gets priority. In other words, if both IRQ-1 and IRQ-4 are requesting attention at the same time, IRQ-1 gets priority and will be triggered first as far as the CPU is concerned. IRQ-4 has to wait until after IRQ-1 has completed its "Interrupt Service Routine" or ISR.

If the opposite happens however, with IRQ-4 doing its ISR (remember, this is software, just like any computer program you might normally write as a computer application), IRQ-1 will "interrupt" the ISR for IRQ-4 and push through its own ISR to be run instead, returning to the IRQ-4 ISR when it has finished. There are exceptions to this as well, but let's keep things simple at the moment.

Let's return for a minute to the original IBM-PC. When it was built, there was only one 8259 chip on the motherboard. When the IBM-AT came out the engineers at IBM decided to add a second 8259 chip to add some additional IRQ signals. Since there was still only 1 pin on the CPU (at this point the 80286) that could receive notification of an interrupt, it was decided to grab IRQ-2 from the original 8259 chip and use that to chain onto the next chip. IRQ-2 was re-routed to IRQ-9 as far as any devices that depended on IRQ-2. The nice thing about going with this scheme was that software that planned on something using IRQ-2 would still be "notified" when that device was used, even though seven other devices were now "sharing" this interrupt. These are IRQ-8 through IRQ-15.

What this means in terms of priorities, however, is that IRQ-8 through IRQ-15 have a higher priority than IRQ-3. This is mainly of concern when you are trying to sort out which device can take precedence over another, and how important it would be to notified when a piece of equipment is trying to get your attention. If you are dealing with software running a specific computer configuration, this priority level is very important.

It should be noted here that COM1 (serial communication channel one) usually uses IRQ-4, and COM2 uses IRQ-3, which has the net effect of making COM2 to be a higher priority for receiving data over COM1. Usually the software really doesn't care, but on some rare occasions you really need to know this fact.

3.4.1 8259 Registers

The 8259 has several "registers" that are associated with I/O port addresses. We will visit this concept a little bit more when we get to the 8250 chip. For a typical PC Computer system, the following are typical primary port addresses associated with the 8259:

Interrupt Controller Port I/O Addresses

Register Name	**I/O Port**
Master Interrupt Controller	$0020
Slave Interrupt Controller	$00A0

This primary port address is what we will use to directly communicate with the 8259 chip in our software. There are a number of commands that can be sent to this chip through these I/O port addresses, but for our purposes we really don't need to deal with them. Most of these are used to do the initial setup and configuration of the computer equipment by the Basic Input Output System (BIOS) of the computer, and unless you are rewriting the BIOS from scratch, you really don't have to worry about this. Also, each computer is a little different in its behavior when you are dealing with equipment at this level, so this is something more for a computer manufacturer to worry about rather than something an application programmer should have to deal with, which is exactly why BIOS software is written at all.

Keep in mind that this is the "typical" Port I/O address for most PC-compatible type computer systems, and can vary depending on what the manufacturer is trying to accomplish. Generally you don't have to worry about incompatibility at this level, but when we get to Port I/O addresses for the serial ports this will become a much larger issue.

3.4.2 Device Registers

I'm going to spend a little time here to explain the meaning of the word register. When you are working with equipment at this level, the electrical engineers who designed the equipment refer to registers that change the configuration of the equipment. This can happen at several levels of abstraction, so I want to clear up some of the confusion.

A register is simply a small piece of RAM that is available for a device to directly manipulate. In a CPU like the 8086 or a Pentium, these are the memory areas that are used to directly perform mathematical operations like adding two numbers together. These usually go by names like AX, SP, etc. There are very few registers on a typical CPU because access to these registers is encoded directly into the basic machine-level instructions.

When we are talking about device register, keep in mind these are not the CPU registers, but instead memory areas on the devices themselves. These are often designed so they are connected to the Port I/O memory, so when you write to or read from the Port I/O addresses, you are directly accessing the device registers. Sometimes there will be a further level of abstraction, where you will have one Port I/O address that will indicate which register you are changing, and another Port I/O address that has the data you are sending to that register. How you deal with the device is based on how complex it is and what you are going to be doing.

In a real sense, they are registers, but keep in mind that often each of these devices can be considered a full computer in its own right, and all you are doing is establishing how it will be communicating with the main CPU. Don't get hung up here and get these confused with the CPU registers.

3.4.3 ISR Cleanup

One area that you have to interact on a regular basis when using interrupt controllers is to inform the 8259 PIC controller that the interrupt service routine is completed. When your software is performing an interrupt handler, there is no automated method for the CPU to signal to the 8259 chip that you have finished, so a specific "register" in the PIC needs to be set to let the next interrupt handler be able to access the computer system. Typical software to accomplish this is like the following:

```
Port[$20] := $20;
```

This is sending the command called "End of Interrupt" or often written as an abbreviation simply "EOI". There are other commands that can be sent to this register, but for our purposes this is the only one that we need to concern ourselves with.

Now this will clear the "master" PIC, but if you are using a device that is triggered on the "slave" PIC, you also need to inform that chip as well that the interrupt service has been completed. This means you need to send "EOI" to that chip as well in a manner like this:

```
Port[$A0] := $20;
Port[$20] := $20;
```

There are other things you can do to make your computer system work smoothly, but let's keep things simple for now.

3.4.4 PIC Device Masking

Before we leave the subject of the 8259 PIC, I'd like to cover the concept of device masking. Each one of the devices that are attached to the PIC can be "turned on" or "turned off" from the viewpoint of how they can interrupt the CPU through the PIC chip. Usually as an application developer all we really care about is if the device is turned on, although if you are trying to isolate performance issues you might turn off some other devices. Keep in mind that if you turn a device "off", the interrupt will not work until it is turned back on. That can include the keyboard or other critical devices you may need to operate your computer.

The register to set this mask is called "Operation Control Word 1" or "OCW1". This is located at the PIC base address + 1, or for the "Master" PIC at Port I/O Address $21. This is where you need to go over bit manipulation, which I won't cover in detail here. The following tables show the related bits to change in order to enable or disable each of the hardware interrupt devices:

Master OCW1 ($21)

Bit	**IRQ Enabled**	**Device Function**
7	IRQ7	Parallel Port (LPT1)
6	IRQ6	Floppy Disk Controller
5	IRQ5	Reserved/Sound Card
4	IRQ4	Serial Port (COM1)
3	IRQ3	Serial Port (COM2)

Master OCW1 ($21)

Bit	IRQ Enabled	Device Function
2	IRQ2	Slave PIC
1	IRQ1	Keyboard
0	IRQ0	System Timer

Slave OCW1 ($A1)

Bit	IRQ Enabled	Device Function
7	IRQ15	Reserved
6	IRQ14	Hard Disk Drive
5	IRQ13	Math Co-Processor
4	IRQ12	PS/2 Mouse
3	IRQ11	PCI Devices
2	IRQ10	PCI Devices
1	IRQ9	Redirected IRQ2 Devices
0	IRQ8	Real Time Clock

Assuming that we want to turn on IRQ3 (typical for the serial port COM2), we would use the following software:

```
Port[$21] := Port[$21] and $F7; {Clearing bit 3 for enabling IRQ3}
```

And to turn it off we would use the following software:

```
Port[$21] := Port[$21] or $08; {Setting bit 3 for disabling IRQ3}
```

If you are having problems getting anything to work, you can simply send this command in your software:

```
Port[$21] := 0;
```

which will simply enable everything. This may not be a good thing to do, but will have to be something for you to experiment with depending on what you are working with. Try not to take short cuts like this as not only is it a sign of a lazy programmer, but it can have side effects that your computer may behave different than you intended. If you are working with the computer at this level, the goal is to change as little as possible so you don't cause damage to any other software you are using.

3.5 Serial COM Port Memory and I/O Allocation

Now that we have pushed through the 8259 chip, lets move on to the UART itself. While the Port I/O addresses for the PICs are fairly standard, it is common for computer manufacturers to move stuff around for the serial ports themselves. Also, if you have serial port devices that are part of an add-in card (like an ISA or PCI card in the expansion slots of your computer), these will usually have different settings than something built into the main

motherboard of your computer. It may take some time to hunt down these settings, and it is important to know what these values are when you are trying to write your software. Often these values can be found in the BIOS setup screens of your computer, or if you can pause the messages when your computer turns on, they can be found as a part of the boot process of your computer.

For a "typical" PC system, the following are the Port I/O addresses and IRQs for each serial COM port:

Common UART IRQ and I/O Port Addresses

COM Port	**IRQ**	**Base Port I/O address**
COM1	IRQ4	$3F8
COM2	IRQ3	$2F8
COM3	IRQ4	$3E8
COM4	IRQ3	$2E8

If you notice something interesting here, you can see that COM3 and COM1 share the same interrupt. This is not a mistake but something you need to keep in mind when you are writing an interrupt service routine. The 15 interrupts that were made available through the 8259 PIC chips still have not been enough to allow all of the devices that are found on a modern computer to have their own separate hardware interrupt, so in this case you will need to learn how to share the interrupt with other devices. I'll cover more of that later when we get into the actual software to access the serial data ports, but for now remember not to write your software strictly for one device.

The Base Port I/O address is important for the next topic we will cover, which is directly accessing the UART registers.

3.6 UART Registers

The UART chip has a total of 12 different registers that are mapped into 8 different Port I/O locations. Yes, you read that correct, 12 registers in 8 locations. Obviously that means there is more than one register that uses the same Port I/O location, and affects how the UART can be configured. In reality, two of the registers are really the same one but in a different context, as the Port I/O address that you transmit the characters to be sent out of the serial data port is the same address that you can read in the characters that are sent to the computer. Another I/O port address has a different context when you write data to it than when you read data from it... and the number will be different after writing the data to it than when you read data from it. More on that in a little bit.

One of the issues that came up when this chip was originally being designed was that the designer needed to be able to send information about the baud rate of the serial data with 16 bits. This actually takes up two different "registers" and is toggled by what is called the "Divisor Latch Access Bit" or "DLAB". When the DLAB is set to "1", the baud rate registers can be set and when it is "0" the registers have a different context.

Does all this sound confusing? It can be, but lets take it one simple little piece at a time. The following is a table of each of the registers that can be found in a typical UART chip:

UART Registers

Base Address	DLAB	I/O Access	Abbrv.	Register Name
+0	0	Write	THR	Transmitter Holding Buffer
+0	0	Read	RBR	Receiver Buffer
+0	1	Read/Write	DLL	Divisor Latch Low Byte
+1	0	Read/Write	IER	Interrupt Enable Register
+1	1	Read/Write	DLH	Divisor Latch High Byte
+2	x	Read	IIR	Interrupt Identification Register
+2	x	Write	FCR	FIFO Control Register
+3	x	Read/Write	LCR	Line Control Register
+4	x	Read/Write	MCR	Modem Control Register
+5	x	Read	LSR	Line Status Register
+6	x	Read	MSR	Modem Status Register
+7	x	Read/Write	SR	Scratch Register

The "x" in the DLAB column means that the status of the DLAB has no effect on what register is going to be accessed for that offset range. Notice also that some registers are Read only. If you attempt to write data to them, you may end up with either some problems with the modem (worst case), or the data will simply be ignored (typically the result). As mentioned earlier, some registers share a Port I/O address where one register will be used when you write data to it and another register will be used to retrieve data from the same address.

Each serial communication port will have its own set of these registers. For example, if you wanted to access the Line Status Register (LSR) for COM1, and assuming the base I/O Port address of $3F8, the I/O Port address to get the information in this register would be found at $3F8 + $05 or $3FD. Some example code would be like this:

```
const
  COM1_Base = $3F8;
  COM2_Base = $2F8;
  LSR_Offset = $05;

function LSR_Value: Byte;
begin
  Result := Port[COM1_Base+LSR_Offset];
end;
```

There is quite a bit of information packed into each of these registers, and the following is an explanation for the meaning of each register and the information it contains.

3.6.1 Transmitter Holding Buffer/Receiver Buffer

Offset: +0 . The Transmit and Receive buffers are related, and often even use the very same memory. This is also one of the areas where later versions of the 8250 chip have a significant

impact, as the later models incorporate some internal buffering of the data within the chip before it gets transmitted as serial data. The base 8250 chip can only receive one byte at a time, while later chips like the 16550 chip will hold up to 16 bytes either to transmit or to receive (sometimes both... depending on the manufacturer) before you have to wait for the character to be sent. This can be useful in multi-tasking environments where you have a computer doing many things, and it may be a couple of milliseconds before you get back to dealing with serial data flow.

These registers really are the "heart" of serial data communication, and how data is transferred from your software to another computer and how it gets data from other devices. Reading and Writing to these registers is simply a matter of accessing the Port I/O address for the respective UART.

3.6.2 Divisor Latch Bytes

Offset: +0 and +1 . The Divisor Latch Bytes are what control the baud rate of the modem. As you might guess from the name of this register, it is used as a divisor to determine what baud rate that the chip is going to be transmitting at.

In reality, it is even simpler than that. This is really a count-down clock that is used each time a bit is transmitted by the UART. Each time a bit is sent, a count-down register is reset to this value and then counts down to zero. This clock is running typically at 115.2 KHz. In other words, at 115 thousand times per second a counter is going down to determine when to send the next bit. At one time during the design process it was anticipated that some other frequencies might be used to get a UART working, but with the large amount of software already written for this chip this frequency is pretty much standard for almost all UART chips used on a PC platform. They may use a faster clock in some portion (like a 1.843 MHz clock), but some fraction of that frequency will then be used to scale down to a 115.2 KHz clock.

Some more on UART clock speeds (advanced coverage): For many UART chips, the clock frequency that is driving the UART is 1.8432 MHz. This frequency is then put through a divider circuit that drops the frequency down by a factor of 16, giving us the 115.2 KHz frequency mentioned above. If you are doing some custom equipment using this chip, the National Semiconductor spec sheets allow for a 3.072 MHz clock and 18.432 MHz clock. These higher frequencies will allow you to communicate at higher baud rates, but require custom circuits on the motherboard and often new drivers in order to deal with these new frequencies. What is interesting is that you can still operate at 50 baud with these higher clock frequencies, but at the time the original IBM-PC/XT was manufactured this wasn't a big concern as it is now for higher data throughput.

If you use the following mathematical formula, you can determine what numbers you need to put into the Divisor Latch Bytes:

$$DivisorLatchValue = \frac{115200}{BaudRate}$$

That gives you the following table that can be used to determine common baud rates for serial communication:

Divisor Latch Byte Values (common baud rates)

Baud Rate	Divisor (in decimal)	Divisor Latch High Byte	Divisor Latch Low Byte
50	2304	$09	$00
110	1047	$04	$17
220	524	$02	$0C
300	384	$01	$80
600	192	$00	$C0
1200	96	$00	$60
2400	48	$00	$30
4800	24	$00	$18
9600	12	$00	$0C
19200	6	$00	$06
38400	3	$00	$03
57600	2	$00	$02
115200	1	$00	$01

One thing to keep in mind when looking at the table is that baud rates 600 and above all set the Divisor Latch High Byte to zero. A sloppy programmer might try to skip setting the high byte, assuming that nobody would deal with such low baud rates, but this is not something to always presume. Good programming habits suggest you should still try to set this to zero even if all you are doing is running at higher baud rates.

Another thing to notice is that there are other potential baud rates other than the standard ones listed above. While this is not encouraged for a typical application, it would be something fun to experiment with. Also, you can attempt to communicate with older equipment in this fashion where a standard API library might not allow a specific baud rate that should be compatible. This should demonstrate why knowledge of these chips at this level is still very useful.

When working with these registers, also remember that these are the only ones that require the Divisor Latch Access Bit to be set to "1". More on that below, but I'd like to mention that it would be useful for application software setting the baud rate to set the DLAB to "1" just for the immediate operation of changing the baud rate, then putting it back to "0" as the very next step before you do any more I/O access to the modem. This is just a good working habit, and keeps the rest of the software you need to write for accessing the UART much cleaner and easier.

One word of caution: Do not set the value "0" for both Divisor Latch bytes. While it will not (likely) damage the UART chip, the behavior on how the UART will be transmitting serial data will be unpredictable, and will change from one computer to the next, or even from one time you boot the computer to the next. This is an error condition, and if you are writing software that works with baud rate settings on this level you should catch potential "0" values for the Divisor Latch.

Here is some sample software to set and retrieve the baud rate for COM1:

```
const
  COM1_Base = $3F8;
  COM2_Base = $2F8;
```

```
  LCR_Offset = $03;
  Latch_Low = $00;
  Latch_High = $01;

procedure SetBaudRate(NewRate: Word);
var
  DivisorLatch: Word;
begin
  DivisorLatch := 115200 div NewRate;
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] or
 $80; {Set DLAB}
  Port[COM1_Base + Latch_High] := DivisorLatch shr 8;
  Port[COM1_Base + Latch_Low] := DivisorLatch and $FF;
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] and
 $7F; {Clear DLAB}
end;

function GetBaudRate: Integer;
var
  DivisorLatch: Word;
begin
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] or
 $80; {Set DLAB}
  DivisorLatch := (Port[COM1_Base + Latch_High] shl 8) +
 Port[COM1_Base + Latch_Low];
  Port[COM1_Base + LCR_Offset] := Port[COM1_Base + LCR_Offset] and
 $7F; {Clear DLAB}
  Result := 115200 div DivisorLatch;
end;
```

3.6.3 Interrupt Enable Register

Offset: +1 . This register allows you to control when and how the UART is going to trigger an interrupt event with the hardware interrupt associated with the serial COM port. If used properly, this can enable an efficient use of system resources and allow you to react to information being sent across a serial data line in essentially real-time conditions. Some more on that will be covered later, but the point here is that you can use the UART to let you know exactly when you need to extract some data. This register has both read- and write-access.

The following is a table showing each bit in this register and what events that it will enable to allow you check on the status of this chip:

Interrupt Enable Register (IER)

Bit	Notes
7	Reserved
6	Reserved
5	Enables Low Power Mode (16750)
4	Enables Sleep Mode (16750)
3	Enable Modem Status Interrupt
2	Enable Receiver Line Status Interrupt
1	Enable Transmitter Holding Register Empty Interrupt
0	Enable Received Data Available Interrupt

The Received Data interrupt is a way to let you know that there is some data waiting for you to pull off of the UART. This is probably the one bit that you will use more than the rest, and has more use.

The Transmitter Holding Register Empty Interrupt is to let you know that the output buffer (on more advanced models of the chip like the 16550) has finished sending everything that you pushed into the buffer. This is a way to streamline the data transmission routines so they take up less CPU time.

The Receiver Line Status Interrupt indicates that something in the LSR register has probably changed. This is usually an error condition, and if you are going to write an efficient error handler for the UART that will give plain text descriptions to the end user of your application, this is something you should consider. This is certainly something that takes a bit more advanced knowledge of programming.

The Modem Status Interrupt is to notify you when something changes with an external modem connected to your computer. This can include things like the telephone "bell" ringing (you can simulate this in your software), that you have successfully connected to another modem (Carrier Detect has been turned on), or that somebody has "hung up" the telephone (Carrier Detect has turned off). It can also help you to know if the external modem or data equipment can continue to receive data (Clear to Send). Essentially, this deals with the other wires in the RS-232 standard other than strictly the transmit and receive wires.

The other two modes are strictly for the 16750 chip, and help put the chip into a "low power" state for use on things like a laptop computer or an embedded controller that has a very limited power source like a battery. On earlier chips you should treat these bits as "Reserved", and only put a "0" into them.

3.6.4 Interrupt Identification Register

Offset: +2 . This register is to be used to help identify what the unique characteristics of the UART chip that you are using has. This chip has two uses:

- Identification of why the UART triggered an interrupt.
- Identification of the UART chip itself.

Of these, identification of why the interrupt service routine has been invoked is perhaps the most important.

The following table explains some of the details of this register, and what each bit on it represents:

Interrupt Identification Register (IIR)

Bit	Notes				
	Bit 7	**Bit 6**			
	0	0	No FIFO on chip		
7 and 6	0	1	Reserved condition		
	1	0	FIFO enabled, but not functioning		
	1	1	FIFO enabled		
5	64 Byte FIFO Enabled (16750 only)				
4	Reserved				
	Bit 3	**Bit 2**	**Bit 1**		**Reset Method**
	0	0	0	Modem Status Interrupt	Reading Modem Status Register(MSR)
3, 2 and 1	0	0	1	Transmitter Holding Register Empty Interrupt	Reading Interrupt Identification Register(IIR) or Writing to Transmit Holding Buffer(THR)
	0	1	0	Received Data Available Interrupt	Reading Receive Buffer Register(RBR)
	0	1	1	Receiver Line Status Interrupt	Reading Line Status Register(LSR)
	1	0	0	Reserved	N/A
	1	0	1	Reserved	N/A
	1	1	0	Time-out Interrupt Pending (16550 & later)	Reading Receive Buffer Register(RBR)
	1	1	1	Reserved	N/A
0	Interrupt Pending Flag				

When you are writing an interrupt handler for the 8250 chip (and later), this is the register that you need to look at in order to determine what exactly was the trigger for the interrupt.

As explained earlier, multiple serial communication devices can share the same hardware interrupt. The use of "Bit 0" of this register will let you know (or confirm) that this was indeed the device that caused the interrupt. What you need to do is check on all serial devices (that are in separate port I/O address spaces), and get the contents of this register. Keep in mind that it is at least possible for more than one device to trigger an interrupt at the same time, so when you are doing this scanning of serial devices, make sure you examine all of them, even one of the first devices did in fact need to be processed. Some computer systems may not require this to occur, but this is a good programming practice anyway. It is also possible that due to how you processed the UARTs earlier, that you have already dealt with all of the UARTs for a given interrupt. When this bit is a "0", it identifies that the UART is triggering an interrupt. When it is "1", that means the interrupt has already been processed or this particular UART was not the triggering device. I know that this seems a little bit backward for a typical bit-flag used in computers, but this is called digital logic being asserted low, and is fairly common with electrical circuit design. This is a bit more unusual through for this logic pattern to go into the software domain.

Bits 1, 2 & 3 help to identify exactly what sort of interrupt event was used within the UART to invoke the hardware interrupt. These are the same interrupts that were earlier enabled with the IER register. In this case, however, each time you process the registers and deal with the interrupt it will be unique. If multiple "triggers" occur for the UART due to many things happening at the same time, this will be invoked through multiple hardware interrupts. Earlier chip sets don't use bit 3, but this is a reserved bit on those UART systems and always set to logic state "0", so programming logic doesn't have to be different when trying to decipher which interrupt has been used.

To explain the FIFO timeout Interrupt, this is a way to check for the end of a packet or if the incoming data stream has stopped. Generally the following conditions must exist for this interrupt to be triggered: Some data needs to be in the incoming FIFO and has not been read by the computer. Data transmissions being sent to the UART via serial data link must have ended with no new characters being received. The CPU processing incoming data must not have retrieved any data from the FIFO before the timeout has occurred. The timeout will occur usually after the period it would take to transmit or receive at least 4 characters. If you are talking about data sent at 1200 baud, 8 data bits, 2 stop bits, odd parity, that would take about 40 milliseconds, which is almost an eternity in terms of things that your computer can accomplish on a 4 GHz Pentium CPU.

The "Reset Method" listed above describes how the UART is notified that a given interrupt has been processed. When you access the register mentioned under the reset method, this will clear the interrupt condition for that UART. If multiple interrupts for the same UART have been triggered, either it won't clear the interrupt signal on the CPU (triggering a new hardware interrupt when you are done), or if you check back to this register (IIR) and query the Interrupt Pending Flag to see if there are more interrupts to process, you can move on and attempt to resolve any new interrupt issue that you may have to deal with, using appropriate application code.

Bits 5, 6 & 7 are reporting the current status of FIFO buffers being used for transmitting and receiving characters. There was a bug in the original 16550 chip design when it was

first released that had a serious flaw in the FIFO, causing the FIFO to report that it was working but in fact it wasn't. Because some software had already been written to work with the FIFO, this bit (Bit 7 of this register) was kept, but Bit 6 was added to confirm that the FIFO was in fact working correctly, in case some new software wanted to ignore the hardware FIFO on the earlier versions of the 16550 chip. This pattern has been kept on future versions of this chip as well. On the 16750 chip an added 64-byte FIFO has been implemented, and Bit 5 is used to designate the presence of this extended buffer. These FIFO buffers can be turned on and off using registers listed below.

3.6.5 FIFO Control Register

Offset: +2 . This is a relatively "new" register that was not a part of the original 8250 UART implementation. The purpose of this register is to control how the First In/First Out (FIFO) buffers will behave on the chip and to help you fine-tune their performance in your application. This even gives you the ability to "turn on" or "turn off" the FIFO.

Keep in mind that this is a "write only" register. Attempting to read in the contents will only give you the Interrupt Identification Register (IIR), which has a totally different context.

FIFO Control Register (FCR)

Bit	**Notes**			
	Bit 7	**Bit 6**	**Interrupt Trigger Level (16 byte)**	**Trigger Level (64 byte)**
7 & 6	0	0	1 Byte	1 Byte
	0	1	4 Bytes	16 Bytes
	1	0	8 Bytes	32 Bytes
	1	1	14 Bytes	56 Bytes
5	Enable 64 Byte FIFO (16750)			
4	Reserved			
3	DMA Mode Select			
2	Clear Transmit FIFO			
1	Clear Receive FIFO			
0	Enable FIFOs			

Writing a "0" to bit 0 will disable the FIFOs, in essence turning the UART into 8250 compatibility mode. In effect this also renders the rest of the settings in this register to become useless. If you write a "0" here it will also stop the FIFOs from sending or receiving data, so any data that is sent through the serial data port may be scrambled after this setting has been changed. It would be recommended to disable FIFOs only if you are trying to reset the serial communication protocol and clearing any working buffers you may have in your application software. Some documentation suggests that setting this bit to "0" also clears the FIFO buffers, but I would recommend explicit buffer clearing instead using bits 1 and 2.

Bits 1 and 2 are used to clear the internal FIFO buffers. This is useful when you are first starting up an application where you might want to clear out any data that may have been "left behind" by a previous piece of software using the UART, or if you want to reset a communications connection. These bits are "automatically" reset, so if you set either of

these to a logical "1" state you will not have to go and put them back to "0" later. Sending a logical "0" only tells the UART not to reset the FIFO buffers, even if other aspects of FIFO control are going to be changed.

Bit 3 is in reference to how the DMA (Direct Memory Access) takes place, primarily when you are trying to retrieve data from the FIFO. This would be useful primarily to a chip designer who is trying to directly access the serial data, and store this data in an internal buffer. There are two digital logic pins on the UART chip itself labeled RXRDY and TXRDY. If you are trying to design a computer circuit with the UART chip this may be useful or even important, but for the purposes of an application developer on a PC system it is of little use and you can safely ignore it.

Bit 5 allows the 16750 UART chip to expand the buffers from 16 bytes to 64 bytes. Not only does this affect the size of the buffer, but it also controls the size of the trigger threshold, as described next. On earlier chip types this is a reserved bit and should be kept in a logical "0" state. On the 16750 it make that UART perform more like the 16550 with only a 16 byte FIFO.

Bits 6 and 7 describe the trigger threshold value. This is the number of characters that would be stored in the FIFO before an interrupt is triggered that will let you know data should be removed from the FIFO. If you anticipate that large amounts of data will be sent over the serial data link, you might want to increase the size of the buffer. The reason why the maximum value for the trigger is less than the size of the FIFO buffer is because it may take a little while for some software to access the UART and retrieve the data. Remember that when the FIFO is full, you will start to lose data from the FIFO, so it is important to make sure you have retrieved the data once this threshold has been reached. If you are encountering software timing problems in trying to retrieve the UART data, you might want to lower the threshold value. At the extreme end where the threshold is set to 1 byte, it will act essentially like the basic 8250, but with the added reliability that some characters may get caught in the buffer in situations where you don't have a chance to get all of them immediately.

3.6.6 Line Control Register

Offset: +3 . This register has two major purposes:

- Setting the Divisor Latch Access Bit (DLAB), allowing you to set the values of the Divisor Latch Bytes.
- Setting the bit patterns that will be used for both receiving and transmitting the serial data. In other words, the serial data protocol you will be using (8-1-None, 5-2-Even, etc.).

Line Control Register (LCR)

Bit	**Notes**			
7	Divisor Latch Access Bit			
6	Set Break Enable			
3, 4 & 5	**Bit 5**	**Bit 4**	**Bit 3**	**Parity Select**

Line Control Register (LCR)

Bit	Notes		
0	0	0	No Parity
0	0	1	Odd Parity
0	1	1	Even Parity
1	0	1	Mark
1	1	1	Space
2	0	One Stop Bit	
1	1.5 Stop Bits or 2 Stop Bits		
0 & 1	**Bit 1**	**Bit 0**	**Word Length**
0	0	5 Bits	
0	1	6 Bits	
1	0	7 Bits	
1	1	8 Bits	

The first two bits (Bit 0 and Bit 1) control how many data bits are sent for each data "word" that is transmitted via serial protocol. For most serial data transmission, this will be 8 bits, but you will find some of the earlier protocols and older equipment that will require fewer data bits. For example, some military encryption equipment only uses 5 data bits per serial "word", as did some TELEX equipment. Early ASCII teletype terminals only used 7 data bits, and indeed this heritage has been preserved with SMTP format that only uses 7-bit ASCII for e-mail messages. Clearly this is something that needs to be established before you are able to successfully complete message transmission using RS-232 protocol.

Bit 2 controls how many stop bits are transmitted by the UART to the receiving device. This is selectable as either one or two stop bits, with a logical "0" representing 1 stop bit and "1" representing 2 stop bits. In the case of 5 data bits, the RS-232 protocol instead sends out "1.5 stop bits". What this means is that one serial data "word" is transmitted with only 1 stop bit, and then the next one is transmitted with 2 stop bits.

Another thing to keep in mind is that the RS-232 standard only specifies that at least one data bit cycle will be kept a logical "1" at the end of each serial data word (in other words, a complete character from start bit, data bits, parity bits, and stop bits). If you are having timing problems between the two computers but are able to in general get the character sent across one at a time, you might want to add a second stop bit instead of reducing baud rate. This adds a one-bit penalty to the transmission speed per character instead of halving the transmission speed by dropping the baud rate (usually).

Bits 3, 4, and 5 control how each serial word responds to parity information. When Bit 3 is a logical "0", this causes no parity bits to be sent out with the serial data word. Instead it moves on immediately to the stop bits, and is an admission that parity checking at this level is really useless. You might still gain a little more reliability with data transmission by including the parity bits, but there are other more reliable and practical ways that will be discussed in other chapters in this book. If you want to include parity checking, the following explains each parity method other than "none" parity:

Odd Parity

Each bit the data portion of the serial word is added as a simple count of the number of logical "1" bits. If this is an odd number of bits, the parity bit will be transmitted as a logical "1".

Even Parity

Like Odd Parity, the bits are added together. In this case, however, if the number of bits end up as an even number it will display as a logical "1", which is the exact opposite of odd parity.

Mark Parity

In this case the parity bit will always be a logical "1". While this may seem a little unusual, this is put in for testing and diagnostics purposes. If you want to make sure that the software on the receiving end of the serial connection is responding correctly to a parity error, you can send a Mark or a Space parity, and send characters that don't meet what the receiving UART or device is expecting for parity. In addition for Mark Parity only, you can use this bit as an extra "stop bit". Keep in mind that RS-232 standards are expecting a logical "1" to end a serial data word, so a receiving computer will not be able to tell the difference between a "Mark" parity bit and a stop bit. In essence, you can have 3 or 2.5 stop bits through the use of this setting and by appropriate use of the stop bit portion of this register as well. This is a way to "tweak" the settings on your computer in a way that typical applications don't allow you to do, or at least gain a deeper insight into serial data settings.

Space Parity

Like the Mark parity, this makes the parity bit "sticky", so it doesn't change. In this case it puts in a logical "0" for the parity bit every time you transmit a character. There are not many practical uses for doing this other than a crude way to put in 9 data bits for each serial word, or for diagnostics purposes.

3.6.7 Modem Control Register

Offset: +4 . This register allows you to do "hardware" flow control, under software control. Or in a more practical manner, it allows direct manipulation of four different wires on the UART that you can set to any series of independent logical states, and be able to offer control of the modem. It should also be noted that most UARTs need Auxiliary Output 2 set to a logical "1" to enable interrupts.

Modem Control Register (MCR)

Bit	**Notes**
7	Reserved
6	Reserved
5	Autoflow Control Enabled (16750)
4	Loopback Mode
3	Auxiliary Output 2
2	Auxiliary Output 1
1	Request To Send
0	Data Terminal Ready

Of these outputs on a typical PC platform, only the Request to Send (RTS) and Data Terminal Ready (DTR) are actually connected to the output of the PC on the DB-9 connector. If you are fortunate to have a DB-25 serial connector (more commonly used for parallel communications on a PC platform), or if you have a custom UART on an expansion card, the auxiliary outputs might be connected to the RS-232 connection. If you are using this chip as a component on a custom circuit, this would give you some "free" extra output signals you can use in your chip design to signal anything you might want to have triggered by a TTL output, and would be under software control. There are easier ways to do this, but in this case it might save you an extra chip on your layout.

The "loopback" mode is primarily a way to test the UART to verify that the circuits are working between your main CPU and the UART. This seldom, if ever, needs to be tested by an end user, but might be useful for some initial testing of some software that uses the UART. When this is set to a logical state of "1", any character that gets put into the transmit register will immediately be found in the receive register of the UART. Other logical signals like the RTS and DTS listed above will show up in the modem status register just as if you had put a loopback RS-232 device on the end of your serial communication port. In short, this allows you to do a loopback test using just software. Except for these diagnostics purposes and for some early development testing of software using the UART, this will never be used.

On the 16750 there is a special mode that can be invoked using the Modem Control Register. Basically this allows the UART to directly control the state of the RTS and DTS for hardware character flow control, depending on the current state of the FIFO. This behavior is also affected by the status of Bit 5 of the FIFO Control Register (FCR). While this is useful, and can change some of the logic on how you would write UART control software, the 16750 is comparatively new as a chip and not commonly found on many computer systems. If you know your computer has a 16750 UART, have fun taking advantage of this increased functionality.

3.6.8 Line Status Register

Offset: +5 . This register is used primarily to give you information on possible error conditions that may exist within the UART, based on the data that has been received. Keep in mind that this is a "read only" register, and any data written to this register is likely to be ignored or worse, cause different behavior in the UART. There are several uses for this information, and some information will be given below on how it can be useful for diagnosing problems with your serial data connection:

Line Status Register (LSR)

Bit	**Notes**
7	Error in Received FIFO
6	Empty Data Holding Registers
5	Empty Transmitter Holding Register
4	Break Interrupt
3	Framing Error
2	Parity Error
1	Overrun Error

Line Status Register (LSR)	
Bit	**Notes**
0	Data Ready

Bit 7 refers to errors that are with characters in the FIFO. If any character that is currently in the FIFO has had one of the other error messages listed here (like a framing error, parity error, etc.), this is reminding you that the FIFO needs to be cleared as the character data in the FIFO is unreliable and has one or more errors. On UART chips without a FIFO this is a reserved bit field.

Bits 5 and 6 refer to the condition of the character transmitter circuits and can help you to identify if the UART is ready to accept another character. Bit 6 is set to a logical "1" if all characters have been transmitted (including the FIFO, if active), and the "shift register" is done transmitting as well. This shift register is an internal memory block within the UART that grabs data from the Transmitter Holding Buffer (THB) or the FIFO and is the circuitry that does the actual transformation of the data to a serial format, sending out one bit of the data at a time and "shifting" the contents of the shift register down one bit to get the value of the next bit. Bit 5 merely tells you that the UART is capable of receiving more characters, including into the FIFO for transmitting.

The Break Interrupt (Bit 4) gets to a logical state of "1" when the serial data input line has not received any new bits for a period of time that is at least as long as an entire serial data "word", including the start bit, data bits, parity bit, and stop bits, for the given baud rate in the Divisor Latch Bytes. Usually this means that the device that is sending serial data to your computer has stopped for some reason. Often with serial communications this is a normal condition, but in this way you have a way to monitor just how the other device is functioning.

Framing errors (Bit 3) occur when the last bit is not a stop bit. Or to be more precise the stop bit is a logical "0". There are several causes for this, including that you have the timing between the two computer mismatched. This is usually caused by a mismatch in baud rate, although other causes might be involved as well, including problems in the physical cabling between the devices or that the cable is too long. You may even have the number of data bits off, so when errors like this are encountered, check the serial data protocol very closely to make sure that all of the settings for the UART (data bit length, parity, and stop bit count) are what should be expected.

Parity errors (Bit 2) can also indicate a mismatched baud rate like the framing errors (particularly if both errors are occurring at the same time). This bit is raised when the parity algorithm that is expected (odd, even, mark, or space) has not been found. If you are using "no parity" in the setup of the UART, this bit should always be a logical "0". When framing errors are not occurring, this is a way to identify that there are some problems with the cabling, although there are other issues you may have to deal with as well.

Overrun errors (Bit 1) are a sign of poor programming or an operating system that is not giving you proper access to the UART. This error condition occurs when there is a character waiting to be read, and the incoming shift register is attempting to move the contents of the next character into the Receiver Buffer (RBR). On UARTs with a FIFO, this also indicates that the FIFO is full as well.

Some things you can do to help get rid of this error including looking at how efficient your software is that is accessing the UART, particularly the part that is monitoring and reading incoming data. On multi-tasking operating systems, you might want to make sure that the portion of the software that reads incoming data is on a separate thread, and that the thread priority is high or time-critical, as this is a very important operation for software that uses serial communications data. A good software practice for applications also includes adding in an application specific "buffer" that is done through software, giving your application more opportunity to be able to deal with the incoming data as necessary, and away from the time critical subroutines needed to get the data off of the UART. This buffer can be as small as 1KB to as large as 1MB, and depends substantially on the kind of data that you are working with. There are other more exotic buffering techniques as well that apply to the realm of application development, and that will be covered in later modules.

If you are working with simpler operating systems like MS-DOS or a real-time operating system, there is a distinction between a poll-driven access to the UART vs. interrupt driven software. Writing an interrupt driver is much more efficient, and there will be a whole section of this book that will go into details of how to write software for UART access.

Finally, when you can't seem to solve the problems of trying to prevent overrun errors from showing up, you might want to think about reducing the baud rate for the serial transmission. This is not always an option, and really should be the option of last choice when trying to resolve this issue in your software. As a quick test to simply verify that the fundamental algorithms are working, you can start with a slower baud rate and gradually go to higher speeds, but that should only be done during the initial development of the software, and not something that gets released to a customer or placed as publicly distributed software.

The Data Ready Bit (Bit 0) is really the simplest part here. This is a way to simply inform you that there is data available for your software to extract from the UART. When this bit is a logical "1", it is time to read the Receiver Buffer (RBR). On UARTs with a FIFO that is active, this bit will remain in a logical "1" state until you have read all of the contents of the FIFO.

3.6.9 Modem Status Register

Offset: +6 . This register is another read-only register that is here to inform your software about the current status of the modem. The modem accessed in this manner can either be an external modem, or an internal modem that uses a UART as an interface to the computer.

Modem Status Register (MSR)

Bit	**Notes**
7	Carrier Detect
6	Ring Indicator
5	Data Set Ready
4	Clear To Send
3	Delta Data Carrier Detect
2	Trailing Edge Ring Indicator
1	Delta Data Set Ready
0	Delta Clear To Send

Bits 7 and 6 are directly related to modem activity. Carrier Detect will stay in a logical state of "1" while the modem is "connect" to another modem. When this goes to a logical state of "0", you can assume that the phone connection has been lost. The Ring Indicator bit is directly tied to the RS-232 wire also labeled "RI" or Ring Indicator. Usually this bit goes to a logical state of "1" as a result of the "ring voltage" on the telephone line is detected, like when a conventional telephone will be ringing to inform you that somebody is trying to call you.

When we get to the section of AT modem commands, there will be other methods that can be shown to inform you about this and other information regarding the status of a modem, and instead this information will be sent as characters in the normal serial data stream instead of special wires. In truth, these extra bits are pretty worthless, but have been a part of the specification from the beginning and comparatively easy for UART designers to implement. It may, however, be a way to efficiently send some additional information or allow a software designer using the UART to get some logical bit signals from other devices for other purposes.

The "Data Set Ready" and "Clear To Send" bits (Bits 4 and 5) are found directly on an RS-232 cable, and are matching wires to "Request To Send" and "Data Terminal Ready" that are transmitted with the "Modem Control Register (MCR). With these four bits in two registers, you can perform "hardware flow control", where you can signal to the other device that it is time to send more data, or to hold back and stop sending data while you are trying to process the information. More will be written about this subject in another module when we get to data flow control.

A note regarding the "delta" bits (Bits 0, 1, 2, and 3). In this case the word "delta" means change, as in a change in the status of one of the bits. This comes from other scientific areas like rocket science where delta-vee means a change in velocity. For the purposes of this register, each of these bits will be a logical "1" the next time you access this Modem Status register if the bit it is associated with (like Delta Data Carrier Detect with Carrier Detect) has changed its logical state from the previous time you accessed this register. The Trailing Edge Ring Indicator is pretty much like the rest, except it is in a logical "1" state only if the "Ring Indicator" bit went from a logical "1" to a logical "0" condition. There really isn't much practical use for this knowledge, but there is some software that tries to take advantage of these bits and perform some manipulation of the data received from the UART based on these bits. If you ignore these 4 bits you can still make a very robust serial communications software.

3.6.10 Scratch Register

Offset: +7 . The Scratch Register is an interesting enigma. So much effort was done to try and squeeze a whole bunch of registers into all of the other I/O port addresses that the designers had an extra "register" that they didn't know what to do with. Keep in mind that when dealing with computer architecture, it is easier when dealing with powers of 2, so they were "stuck" with having to address 8 I/O ports. Allowing another device to use this extra I/O port would make the motherboard design far too complicated.

On some variants of the 8250 UART, any data written to this scratch register will be available to software when you read the I/O port for this register. In effect, this gives you

one extra byte of "memory" that you can use in your applications in any way that you find useful. Other than a virus author (maybe I shouldn't give any ideas), there isn't really a good use for this register. Of limited use is the fact that you can use this register to identify specific variations of the UART because the original 8250 did not store the data sent to it through this register. As that chip is hardly ever used anymore on a PC design (those companies are using more advanced chips like the 16550), you will not find that "bug" in most modern PC-type platforms. More details will be given below on how to identify through software which UART chip is being used in your computer, and for each serial port.

3.7 Software Identification of the UART

Just as it is possible to identify many of the components on a computer system through just software routines, it is also possible to detect which version or variant of the UART that is found on your computer as well. The reason this is possible is because each different version of the UART chip has some unique qualities that if you do a process of elimination you can identify which version you are dealing with. This can be useful information if you are trying to improve performance of the serial I/O routines, know if there are buffers available for transmitting and sending information, as well as simply getting to know the equipment on your PC better.

One example of how you can determine the version of the UART is if the Scratch Register is working or not. On the first 8250 and 8250A chips, there was a flaw in the design of those chip models where the Scratch Register didn't work. If you write some data to this register and it comes back changed, you know that the UART in your computer is one of these two chip models.

Another place to look is with the FIFO control registers. If you set bit "0" of this register to a logical **1**, you are trying to enable the FIFOs on the UART, which are only found in the more recent version of this chip. Reading bits "6" and "7" will help you to determine if you are using either the 16550 or 16550A chip. Bit "5" will help you determine if the chip is the 16750.

Below is a full pseudo code algorithm to help you determine the type of chip you are using:

```
Set the value "0xE7" to the FCR to test the status of the FIFO
flags.
Read the value of the IIR to test for what flags actually got set.
If Bit 6 is set Then
  If Bit 7 is set Then
    If Bit 5 is set Then
      UART is 16750
    Else
      UART is 16550A
    End If
  Else
    UART is 16550
  End If
Else you know the chip doesn't use FIFO, so we need to check the
scratch register
  Set some arbitrary value like 0x2A to the Scratch Register.
  You don't want to use 0xFF or 0x00 as those might be returned by
the Scratch Register instead for a false postive result.
```

```
  Read the value of the Scratch Register
  If the arbitrary value comes back identical
    UART is 16450
  Else
    UART is 8250
  End If
End If
```

When written in Pascal, the above algorithm ends up looking like this:

```
const
  COM1_Addr = $3F8;
  FCR = 2;
  IIR = 2;
  SCR = 7;
```

```
function IdentifyUART: String;
var
  Test: Byte;
begin
  Port[COM1_Addr + FCR] := $E7;
  Test := Port[COM1_Addr + IIR];
  if (Test and $40) > 0 then
    if (Test and $80) > 0 then
      if (Test and $20) > 0 then
        IdentifyUART := '16750'
      else
        IdentifyUART := '16550A'
    else
      IdentifyUART := '16550'
  else begin
    Port[COM1_Addr + SCR] := $2A;
    if Port[COM1_Addr + SCR] = $2A then
      IdentifyUART := '16450'
    else
      IdentifyUART := '8250';
  end;
end;
```

We still havn't identified between the 8250, 8250A, or 8250B; but that is rather pointless anyway on most current computers as it is very unlikely to even find one of those chips because of their age.

A very similar procedure can be used to determine the CPU of a computer, but that is beyond the scope of this book.

3.8 External References

- History of Interrupt Programming[1]
- 8259 Chip Information with other registers explained[2] (dead link?)
- Interfacing the Serial / RS232 Port[3]

1 http://www.cs.clemson.edu/~mark/interrupts.html
2 http://satyap.csoft.net/8259.html
3 http://www.beyondlogic.org/serial/serial.htm

While the 8250 is by far the most popular UART on desktop computers, other popular UARTs include:

- the UART inside the Atmel AVR[4]: ... Embedded_Systems/Atmel_AVR#Serial_-Communication[5]
- the UART inside the Microchip PIC[6]: "Microchip AN774: Asynchronous Communications with the PICmicro® USART"[7]
- the UART inside the Apple Macintosh: ...
- "bit-banging" a UART: ... http://microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012058

3.9 Other Serial Programming Articles

Category:Serial Programming[8]

4 http://en.wikibooks.org/wiki/Embedded_Systems%2FAtmel_AVR
5 http://en.wikibooks.org/wiki/Embedded_Systems%2FAtmel_AVR%23Serial_Communication
6 http://en.wikibooks.org/wiki/PIC
7 http://microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012073
8 http://en.wikibooks.org/wiki/Category%3ASerial%20Programming

4 Serial DOS

4.1 Introduction

It is now time to build on everything that has been established so far. While it is unlikely that you are going to be using MS-DOS for a major application, it is a good operating system to demonstrate a number of ideas related to software access of the 8250 UART and driver development. Compared to modern operating systems like Linux, OS-X, or Windows, MS-DOS can hardly be called an operating system at all. All it really offers is basic access to the hard drive and a few minor utilities. That really doesn't matter so much for what we are dealing with here, and it is a good chance to see how we can directly manipulate the UART to get the full functionality of all aspects of the computer. The tools I'm using are all available for free (as in beer) and can be used in emulator software (like VMware or Bochs) to try these ideas out as well. Emulation of serial devices is generally a weak point for these programs, so it may work easier if you work from a floppy boot of DOS, or on an older computer that is otherwise destined for the trash can because it is obsolete.

For Pascal, you can look here:

- Turbo Pascal `http://bdn.borland.com/article/0,1410,20803,00.html` version 5.5 - This is the software I'm actually using for these examples, and the compiler that most older documentation on the web will also support (generally).
- Free Pascal `http://www.freepascal.org/` - *note* this is a 32-bit version, although there is a port for DOS development. Unlike Turbo Pascal, it also has ongoing development and is more valuable for serious projects running in DOS.

For MS-DOS substitution (if you don't happen to have MS-DOS 6.22 somewhere):

- FreeDOS `http://www.freedos.org/` Project - Now that Microsoft has abandoned development of DOS, this is pretty much the only OS left that is pure command line driven and following the DOS architecture.

4.2 Hello World, Serial Data Version

In the introduction[1], I mentioned that it was very difficult to write computer software that implements RS-232 serial communications. A very short program shows that at least a basic program really isn't that hard at all. In fact, just three more lines than a typical "Hello World" program.

---

1 `http://en.wikibooks.org/wiki/Programming%3ASerial%20Data%20Communications%23Intended%20Audience`

```
program HelloSerial;
var
  DataFile: Text;
begin
  Assign(DataFile,'COM1');
  Rewrite(DataFile);
  Writeln(DataFile,'Hello World');
  Close(DataFile);
end.
```

All of this works because in DOS (and all version of Windows as well... on this particular point) has a "reserved" file name called COM1 that is the operating system hooks into the serial communications ports. While this seems simple, it is deceptively simple. You still don't have access to being able to control the baud rate or any of the other settings for the modem. That is a fairly simple thing to add, however, using the knowledge of the UART discussed in the previous chapter Programming the 8250 UART[2].

To try something even easier, you don't even need a compiler at all. This takes advantage of the reserved "device names" in DOS and can be done from the command prompt.

```
C:\>COPY CON COM1
```

What you are doing here is taking input from *CON* (the console or the standard keyboard you use on your computer) and it "copies" the data to *COM1*. You can also use variations of this to do some interesting file transfers, but it has some important limitations. Most importantly, you don't have access to the UART settings, and this simply uses whatever the default settings of the UART might be, or what you used last time you changed the settings to become with a serial terminal program.

4.3 Finding the Port I/O Address for the UART

The next big task that we have to work with is trying to find the base "address" of the Port I/O so that we can communicate with the UART chip directly (see the part about interface logic in the Typical RS232-Hardware Configuration[3] module for information what this is about). For a "typical" PC system, the following are usually the addresses that you need to work with:

Serial Port Name	Base I/O Port Address	IRQ (interrupt) Number
COM1	3F8	4
COM2	2F8	3
COM3	3E8	4
COM4	2E8	3

2 http://en.wikibooks.org/wiki/Serial%20Programming%3A8250%20UART%20Programming
3 http://en.wikibooks.org/wiki/Serial_Programming%3ATypical_RS232-Hardware_Configuration

4.3.1 Looking up UART Base Address in RAM

We will get back to the issue of the IRQ Number in a little bit, but for now we need to know where to start accessing information about each UART. As demonstrated previously, DOS also keeps track of where the UART IO ports are located at for its own purpose, so you can try to "look up" within the memory tables that DOS uses to try and find the correct address as well. This doesn't always work, because we are going outside of the normal DOS API structure. Alternative operating systems (FreeDOS works fine here) that are otherwise compatible with MS-DOS may not work in this manner, so take note that this may simply give you a wrong result altogether.

The addresses for the serial I/O Ports can be found at the following locations in RAM:

Port	Segment	Offset
COM1	$0040	$0000
COM2	$0040	$0002
COM3	$0040	$0004
COM4	$0040	$0006

Those addresses are written to memory by the BIOS when it boots. If one of the ports doesn't exist, the BIOS writes zero to the respective address. Note that the addresses are given in segment:offset format and that you have to multiply the address of the segment with 16 and add the offset to get to the physical address in memory. This is where DOS "finds" the port addresses so you can run the first sample program in this chapter.

In assembler you can get the addresses like this:

```
; Data Segment
.data
Port  dw 0
...

; Code Segment
.code
mov ax,40h
mov es,ax
mov si,0
mov bx,Port ; 0 - COM1 , 1 - COM2 ...
shl bx,1
mov Port, es:[si+bx]
```

In Turbo Pascal, you can get at these addresses almost the same way and in some ways even easier because it is a "high level language". All you have to do is add the following line to access the COM Port location as a simple array:

```
var
  ComPort: array [1..4] of Word absolute $0040:$0000;
```

The reserved, non standard, word **absolute** is a flag to the compiler that instead of "allocating" memory, that you already have a place in mind to have the computer look instead. This is something that should seldom be done by a programmer unless you are accessing things like these I/O port addresses that are always stored in this memory location.

For a complete program that simply prints out a table of the I/O port addresses for all four standard COM ports, you can use this simple program:

```
program UARTLook;
const
  HexDigits: array [$0..$F] of Char = '0123456789ABCDEF';
var
  ComPort: array [1..4] of Word absolute $0040:$0000;
  Index: Integer;
function HexWord(Number:Word):String;
begin
  HexWord := '$' + HexDigits[Hi(Number) shr 4] +
                   HexDigits[Hi(Number) and $F] +
                   HexDigits[Lo(Number) shr 4] +
                   HexDigits[Lo(Number) and $F];
end;
begin
  writeln('Serial COMport I/O Port addresses:');
  for Index := 1 to 4 do begin
    writeln('COM',Index,' is located at ',HexWord(ComPort[Index]));
  end;
end.
```

4.3.2 Searching BIOS Setup

Assuming that the standard I/O addresses don't seem to be working for your computer and you haven't been able to find the correct I/O Port offset addresses through searching RAM either, all hope is still not lost. Assuming that you have not accidentally changed these settings earlier, you can also try to look up these numbers in the BIOS setup page for your computer. It may take some pushing around to find this information, but if you have a conventional serial data port on your computer, it will be there.

If you are using a serial data port that is connected via USB (common on more recent computers), you are simply not going to be (easily) able to do direct serial data communications in DOS. Instead, you need to use more advanced operating systems like Windows or Linux that is beyond the scope of this chapter. We will cover how to access the serial communications routines in those operating systems in subsequent chapters. The basic principles we are discussing here would still be useful to review because it goes into the basic UART structure.

While it may be useful to try and make IRQs selectable and not presume that the information listed above is correct in all situations, it is important to note that most PC-compatible computer equipment usually has these IRQs and I/O port addresses used in this way because of legacy support. And surprisingly as computers get more sophisticated with even more advanced equipment like USB devices, these legacy connections still work for most equipment.

4.4 Making modifications to UART Registers

Now that we know where to look in memory to modify the UART registers, let's put that knowledge to work. We are also now going to do some practical application of the tables listed earlier in the chapter 8250 UART Programming[4].

To start with, let's redo the previous "Hello World" application, but this time we are going to set the RS-232 transmission parameters to 1200 baud, 7 databits, even parity, and 2 stop bits. I'm choosing this setting parameter because it is not standard for most modem applications, as a demonstration. If you can change these settings, then other transmission settings are going to be trivial.

First, we need to set up some software constants to keep track of locations in memory. This is mainly to keep things clear to somebody trying to make changes to our software in the future, not because the compiler needs it.

```
const
  LCR = 3;
  Latch_Low = $00;
  Latch_High = $01;
```

Next, we need to set the DLAB to a logical "1" so we can set the baud rate:

```
Port[ComPort[1] + LCR] := $80;
```

In this case, we are ignoring the rest of the settings for the Line Control Register (LCR) because we will be setting them up in a little bit. Remember this is just a "quick and dirty" way to get this done for now. A more "formal" way to set up things like baud rate will be demonstrated later on with this module.

Following this, we need to put in the baud rate for the modem. Looking up 1200 baud on the Divisor Latch Bytes table[5] gives us the following values:

```
Port[ComPort[1] + Latch_High] := $00;
Port[ComPort[1] + Latch_Low] := $60;
```

Now we need to set the values for the LCR based on our desired setting of 7-2-E for the communication settings. We also need to "clear" the DLAB which we can also do at the same time.

```
Clearing DLAB = 0 * 128
Clearing "Set Break" flag = 0 * 64
Even Parity = 2 * 8
Two Stop bits = 1 * 4
7 Data bits = 2 * 1
```

4 http://en.wikibooks.org/wiki/Serial%20Programming%3A8250%20UART%20Programming

5 http://en.wikibooks.org/wiki/Serial%20Programming%3A8250%20UART%20Programming%23Divisor%20Latch%20Bytes

```
Port[ComPort[1] + LCR] := $16  {8*2 + 4 + 2 = 22 or $16 in hex}
```

Are things clear so far? What we have just done is some bit-wise arithmetic, and I'm trying to keep things very simple here and to try and explain each step in detail. Let's just put the whole thing together as the quick and dirty "Hello World", but with adjustment of the transmission settings as well:

```
program HelloSerial;
const
  LCR = 3;
  Latch_Low = $00;
  Latch_High = $01;
var
  ComPort: array [1..4] of Word absolute $0040:$0000;
  DataFile: Text;
begin
  Assign(DataFile,'COM1');
  Rewrite(DataFile);
  {Change UART Settings}
  Port[ComPort[1] + LCR] := $80;
  Port[ComPort[1] + Latch_High] := $00;
  Port[ComPort[1] + Latch_Low] := $60;
  Port[ComPort[1] + LCR] := $16
  Writeln(DataFile,'Hello World');
  Close(DataFile);
end.
```

This is getting a little more complicated, but not too much. Still, all we have done so far is just write data out to the serial port. Reading data from the serial data port is going to be a little bit trickier.

4.5 Basic Serial Input

In theory, you could use a standard I/O library and simply read data from the COM port like you would be reading from a file on your hard drive. Something like this:

```
Readln(DataFile,SomeSerialData);
```

There are some problems with doing that with most software, however. One thing to keep in mind is that using a standard input routine will stop your software until the input is finished ending with a "Enter" character (ASCII code 13 or in hex $0D).

Usually what you want to do with a program that receives serial data is to allow the user to do other things while the software is waiting for the data input. In a multitasking operating system, this would simply be put on another "thread", but with this being DOS, we don't (usually) have threading capabilities, nor is it necessary. There are some other alternatives that we do in order to get the serial data brought into your software.

4.5.1 Polling the UART

Perhaps the easiest to go, besides simply letting the standard I/O routines grab the input) is to do software polling of the UART. One of the reasons why this works is because serial communications is generally so slow compared to the CPU speed that you can perform many tasks in between each character being transmitted to your computer. Also, we are trying to do practical applications using the UART chip, so this is a good way to demonstrate some of the capabilities of the chip beyond simple output of data.

Serial Echo Program

Looking at the Line Status Register (LSR), there is a bit called **Data Ready** that indicates there is some data available to your software in the UART. We are going to take advantage of that bit, and start to do data access directly from the UART instead of relying on the standard I/O library. This program we are going to demonstrate here is going to be called *Echo* because all it does is take whatever data is sent to the computer through the serial data port and display it on your screen. We are also going to be configuring the RS-232 settings to a more normal 9600 baud, 8 data bits, 1 stop bit, and no parity. To quit the program, all you have to do is press any key on your keyboard.

```
program SerialEcho;
uses
  Crt;
const
  RBR = 0;
  LCR = 3;
  LSR = 5;
  Latch_Low = $00;
  Latch_High = $01;
var
  ComPort: array [1..4] of Word absolute $0040:$0000;
  InputLetter: Char;
begin
  Writeln('Serial Data Terminal Character Echo Program.  Press any
key on the keyboard to quit.');
  {Change UART Settings}
  Port[ComPort[1] + LCR] := $80;
  Port[ComPort[1] + Latch_High] := $00;
  Port[ComPort[1] + Latch_Low] := $0C;
  Port[ComPort[1] + LCR] := $03;
  {Scan for serial data}
  while not KeyPressed do begin
    if (Port[ComPort[1] + LSR] and $01) > 0 then begin
      InputLetter := Chr(Port[ComPort[1] + RBR]);
      Write(InputLetter);
    end; {if}
  end; {while}
end.
```

Simple Terminal

This program really isn't that complicated. In fact, a very simple "terminal" program can be adapted from this to allow both sending and receiving characters. In this case,

the *Escape* key will be used to quit the program, which will in fact be where most of the changes to the program will happen. We are also introducing for the first time direct output into the UART instead of going through the standard I/O libraries with this line:

```
Port[ComPort[1] + THR] := Ord(OutputLetter);
```

The Transmit Holding Register (THR) is how data you want to transmit gets into the UART in the first place. DOS just took care of the details earlier, so now we don't need to open a "file" in order to send data. We are going to assume, to keep things very simple, that you can't type at 9600 baud, or roughly 11,000 words per minute. Only if you are dealing with very slow baud rates like 110 baud is that going to be an issue anyway (still at over 130 words per minute of typing... a very fast typist indeed).

```
program SimpleTerminal;
uses
  Crt;
const
  THR = 0;
  RBR = 0;
  LCR = 3;
  LSR = 5;
  Latch_Low = $00;
  Latch_High = $01;
  {Character Constants}
  NullLetter = #0;
  EscapeKey = #27;
var
  ComPort: array [1..4] of Word absolute $0040:$0000;
  InputLetter: Char;
  OutputLetter: Char;
begin
  Writeln('Simple Serial Data Terminal Program.  Press "Esc" to
quit.');
  {Change UART Settings}
  Port[ComPort[1] + LCR] := $80;
  Port[ComPort[1] + Latch_High] := $00;
  Port[ComPort[1] + Latch_Low] := $0C;
  Port[ComPort[1] + LCR] := $03;
  {Scan for serial data}
  OutputLetter := NullLetter;
  repeat
    if (Port[ComPort[1] + LSR] and $01) > 0 then begin
      InputLetter := Chr(Port[ComPort[1] + RBR]);
      Write(InputLetter);
    end; {if}
    if KeyPressed then begin
      OutputLetter := ReadKey;
      Port[ComPort[1] + THR] := Ord(OutputLetter);
    end; {if}
  until OutputLetter = EscapeKey;
end.
```

4.6 Interrupt Drivers in DOS

The software polling method may be adequate for most simple tasks, and if you want to test some serial data concepts without writing a lot of software, it may be sufficient. Quite a bit can be done with just that method of data input.

When you are writing a more complete piece of software, however, it becomes important to worry about the efficiency of your software. While the computer is "polling" the UART to see if a character has been sent through the serial communications port, it spends quite a few CPU cycles doing absolutely nothing at all. It also get very difficult to expand a program like the one demonstrated above to become a small section of a very large program. If you want to get that last little bit of CPU performance out of your software, we need to turn to interrupt drivers and how you can write them.

I'll openly admit that this is a tough leap in complexity from a simple polling application listed above, but it is an important programming topic in general. We are also going to expose a little bit about the low-level behavior of the 8086 chip family, which is knowledge you can use in newer operating systems as well, at least for background information.

Going back to earlier discussions about the 8259 Programmable Interrupt Controller (PIC) chip, external devices like the UART can "signal" the 8086 that an important task needs to occur that **interrupts** the flow of the software currently running on the computer. Not all computers do this, however, and sometimes the software polling of devices is the only way to get data input from other devices. The real advantage of interrupt events is that you can process data acquisition from devices like the UART very quickly, and CPU time spent trying to test if there is data available can instead be used for other tasks. It is also useful when designing operating systems that are *event driven*.

Interrupt Requests (IRQs) are labeled with the names IRQ0 to IRQ15. UART chips typically use either IRQ 3 or IRQ 4. When the PIC signals to the CPU that an interrupt has occurred, the CPU automatically start to run a very small subroutine that has been previously setup in the **Interrupt Table** in RAM. The exact routine that is started depends on which IRQ has been triggered. What we are going to demonstrate here is the ability to write our own software that "takes over" from the operating system what should occur when the interrupt occurs. In effect, writing our own "operating system" instead, at least for those parts we are rewriting.

Indeed, this is exactly what operating system authors do when they try to make a new OS... deal with the interrupts and write the subroutines necessary to control the devices connected to the computer.

The following is a very simple program that captures the keyboard interrupt and produces a "clicking" sound in the speaker as you type each key. One interesting thing about this whole section, while it is moving slightly off topic, this is communicating with a serial device. The keyboard on a typical PC transmits the information about each key that you press through a RS-232 serial protocol that operates usually between 300 and 1200 baud and has its own custom UART chip. Normally this isn't something you are going to address, and seldom are you going to have another kind of device connected to the keyboard port, but it is interesting that you can "hack" into the functions of your keyboard by understanding serial data programming.

```
program KeyboardDemo;
uses
  Dos, Crt;
const
  EscapeKey = #27;
var
  OldKeybrdVector: Procedure;
{$F+}
procedure Keyclick; interrupt;
begin
  if Port[$60] < $80 then begin
    Sound(5000);
    Delay(1);
    Nosound;
  end;
  inline($9C) { PUSHF - Push the flags onto the stack }
  OldKeybrdVector;
end;
{$F-}
begin
  GetIntVec($9,@OldKeybrdVector);
  SetIntVec($9,Addr(Keyclick));
  repeat
   if KeyPressed then begin
     OutputLetter := ReadKey;
     Write(OutputLetter);
   end; {if}
  until OutputLetter = EscapeKey;
  SetIntVec($9,@OldKeybrdVector);
end.
```

There are a number of things that this program does, and we need to explore the realm of 16-bit DOS software as well. The 8086 chip designers had to make quite a few compromises in order to work with the computer technology that was available at the time it was designed. Computer memory was quite expensive compared to the overall cost of the computer. Most of the early microcomputers that the IBM-PC was competing against only had 64K or 128K of main CPU RAM anyway, so huge programs were not considered important. In fact, the original IBM-PC was designed to operate on only 128K of RAM although it did become standard with generally up to 640K of main RAM, especially by the time the IBM PC-XT was released and the market for PC "clones" turned out what is generally considered the "standard PC" computer today.

The design came up with what is called **segmented memory**, where the CPU address is made up of a memory "segment" pointer and a 64K block of memory. That is why some early software on these computers could only run in 64K of memory, and created nightmares for compiler authors on the 8086. Pentium computers don't generally have this issue, as the memory model in "protected mode" doesn't use this segmented design methodology.

4.6.1 Far Procedure Calls

```
{$F+}
{$F-}
```

This program has two "compiler switches" that inform the compiler of the need to use what are called far procedure calls. Normally for small programs and simple subroutines, you are

able to use what is called relative indexing with the software so the CPU "jumps" to the portion of RAM with the procedure by doing a bit of simple math and "adding" a number to the current CPU address in order to find the correct instructions. This is done especially because it uses quite a bit less memory to store all of these instructions.

Sometimes, however, a procedure must be accessed from somewhere in RAM that is quite different from the current CPU memory address "instruction pointer". Interrupt procedures are one of these, because it doesn't even have to be the same program that is stored in the interrupt vector table. That brings up the next part to discuss:

4.6.2 Interrupt Procedures

```
procedure Keyclick; interrupt;
```

The word "interrupt" after this procedure name is a key item here. This tells the compiler that it must do something a little bit different when organizing this function than how a normal function call behaves. Typically for most software on the computer, you have a bunch of simple instructions that are then followed by (in assembler) an instruction called:

```
RET
```

This is the mnemonic assembly instruction for return from procedure call. Interrupts are handled a little bit differently and should normally end with a different CPU instruction that in assembly is called:

```
IRET
```

or Interrupt return for short. One of the things that should also happen with any interrupt service routine is to "preserve" the CPU information before doing anything else. Each "command" that you write in your software will modify the internal registers of the CPU. Keep in mind that an interrupt can occur right in the middle of doing some calculations for another program, like rendering a graphic image or making payroll calculations. We need to hand onto that information and "restore" those values on all of the CPU registers at the end of our subroutine. This is usually done by "pushing" all of the register values onto the CPU stack, performing the ISR, and then restoring the CPU registers afterward.

In this case, Turbo Pascal (and other well-written compilers having a compiler flag like this) takes care of these low-level details for you with this simple flag. If the compiler you are using doesn't have this feature, you will have to add these features "by hand" and explicitly put them into your software. That doesn't mean the compiler will do everything for you to make an interrupt procedure. There are more steps to getting this to work still.

4.6.3 Procedure Variables

```
var
  OldKeybrdVector: Procedure;
```

These instructions are using what is called a procedure variable. Keep in mind that all software is located in the same memory as variables and other information your software is using. Essentially, a variable procedure where you don't need to worry about what it does until the software is running, and you can change this variable while your program is running. This is a powerful concept that is not often used, but it can be used for a number of different things. In this case we are keeping track of the previous interrupt service routine and "chaining" these routines together.

There are programs called Terminate and Stay Resident (TSRs) that are loaded into your computer. Some of these are called drivers, and the operating system itself also puts in subroutines to do basic functions. If you want to "play nice" with all of this other software, the established protocol for making sure everybody gets a chance to review the data in an interrupt is to link each new interrupt subroutine to the previously stored interrupt vector. When we are done with whatever we want to do with the interrupt, we then let all of the other programs get a chance to used the interrupt as well. It is also possible that the Interrupt Service Routine (ISR) that we just wrote is not the first one in the chain, but instead one that is being called by another ISR.

4.6.4 Getting/Setting Interrupt Vectors

```
GetIntVec($9,@OldKeybrdVector);
SetIntVec($9,Addr(Keyclick));
SetIntVec($9,@OldKeybrdVector);
```

Again, this is Turbo Pascal "hiding" the details in a convenient way. There is a "vector table" that you can directly access, but this vector table is not always in the same location in RAM. If instead you go through the BIOS with a software interrupt, you are "guaranteed" that the interrupt vector will be correctly replaced.

4.6.5 Hardware Interrupt Table

Interrupt	**Hardware IRQ**	**Purpose**
$00	CPU	Divide by Zero
$01	CPU	Single Step Instruction Processing
$02	CPU	Non-maskable Interrupts
$03	CPU	Breakpoint Instruction
$04	CPU	Overflow Instruction
$05	CPU	Bounds Exception
$06	CPU	Invalid Op Code
$07	CPU	Math Co-processor not found
$08	IRQ0	System Timer
$09	IRQ1	Keyboard
$0A	IRQ2	Cascade from IRQ8 - IRQ15
$0B	IRQ3	Serial Port (COM2)
$0C	IRQ4	Serial Port (COM1)
$0D	IRQ5	Sound Card
$0E	IRQ6	Floppy Disk Controller

Interrupt	**Hardware IRQ**	**Purpose**
$0F	IRQ7	Parallel Port (LPT1)
$10 - $6F		Software Interrupts
$70	IRQ8	Real-time Clock
$71	IRQ9	Legacy IRQ2 Devices
$72	IRQ10	Reserved (often PCI devices)
$73	IRQ11	Reserved (often PCI devices)
$74	IRQ12	PS/2 Mouse
$75	IRQ13	Math Co-Processor Results
$76	IRQ14	Hard Disk Drive
$77	IRQ15	Reserved
$78 - $FF		Software Interrupts

This table gives you a quick glance at some of the things that interrupts are used for, and the interrupt numbers associated with them. Keep in mind that the IRQ numbers are mainly reference numbers, and that the CPU uses a different set of numbers. The keyboard IRQ, for example, is IRQ1, but it is numbered as interrupt $09 inside the CPU.

There are also several interrupts that are "generated" by the CPU itself. While technically hardware interrupts, these are generated by conditions *within* the CPU, sometimes based on conditions setup by your software or the operating system. When we start writing the interrupt service routine for the serial communication ports, we will be using interrupts 11 and 12 ($0B and $0C in hex). As can be seen, most interrupts are assigned for specific tasks. I've omitted the software interrupts mainly to keep this on topic regarding serial programming and hardware interrupts.

4.6.6 Other features

There are several other parts to this program that don't need too much more explanation. Remember, we are talking about serial programming, not interrupt drivers. I/O Port $60 is interesting as this is the Receiver Buffer (RBR) for the keyboard UART. This returns the keyboard "scan code", not the actual character pressed. In fact, when you use a keyboard on a PC, the keyboard actually transmits two characters for each key that you use. One character is transmitted when you press the key down, and another character when the key is "released" to go back up. In this case, the interrupt service routine in DOS normally converts the scan codes into ASCII codes that your software can use. In fact, simple keys like the shift key are treated as just another scan code.

The sound routines access the internal PC speaker, not something on a sound card. About the only thing that uses this speaker any more is the BIOS "beep codes" that you hear only when there is a hardware failure to your computer, or the quick "beep" when you start or reboot the computer. It was never designed for doing things like speech synthesis or music playback, and driver attempts to use it for those purposes sound awful. Still, it is something neat to experiment with and a legacy computer part that is surprisingly still used on many current computers..

4.7 Terminal Program Revisited

I'm going to go back to the serial terminal program for a bit and this time redo the application by using an interrupt service routine. There are a few other concepts I'd like to introduce as well so I'll try to put them in with this example program. From the user perspective, I would like to add the ability to change the terminal characteristics from the command line and allow an "end-user" the ability to change things like the baud rate, stop bits, and parity checking, and allow these to be variables instead of hard-coded constants. I'll explain each section and then put it all together when we are through.

4.7.1 Serial ISR

This is an example of a serial ISR we can use:

```
{$F+}
procedure SerialDataIn; interrupt;
var
 InputLetter: Char;
begin
  if (Port[ComPort[1] + LSR] and $01) > 0 then begin
    InputLetter := Chr(Port[ComPort[1] + RBR]);
 end; {if}
end;
{$F-}
```

This isn't that much different from the polling method that we used earlier, but keep in mind that by placing the checking inside an ISR that the CPU is only doing the check when there is a piece of data available. Why even check the LSR to see if there is a data byte available? Reading data sent to the UART is not the only reason why the UART will invoke an interrupt. We will go over that in detail in a later section, but for now this is good programming practice as well, to confirm that the data is in there.

By moving this checking to the ISR, more CPU time is available for performing other tasks. We could even put the keyboard polling into an ISR as well, but we are going to keep things very simple for now.

4.7.2 FIFO disabling

There is one minor problem with the way we have written this ISR. We are assuming that there is no FIFO in the UART. The "bug" that could happen with this ISR as it is currently written is that multiple characters can be in the FIFO buffer. Normally when this happens, the UART only sends a single interrupt, and it is up to the ISR to "empty" the FIFO buffer completely.

Instead, all we are going to do is simply disable the FIFO completely. This can be done using the FCR (FIFO Control Register) and explicitly disabling the FIFO. As an added precaution, we are also going to "clear" the FIFO buffers in the UART as a part of the initialization portion of the program. Clearing the FIFOs look like this:

```
Port[ComPort[1] + FCR] := $07; {clearing the FIFOs}
```

Disabling the FIFOs look like this:

```
Port[ComPort[1] + FCR] := $00; {disabling FIFOs}
```

We will be using the FIFOs in the next section, so this is more a brief introduction to this register so far.

4.7.3 Working with the PIC

Up until this point, we didn't have to worry about working with the Programmable Interrupt Controller (the PIC). Now we need to. There isn't the need to do all of the potential instructions for the PIC, but we do need to enable and disable the interrupts that are used by the UART. There are two PICs typically on each PC, but due to the typical UART IRQ vector, we really only have to deal with the master PIC.

Pic Function	I/O Port Address
PIC Commands	0x20
Interrupt Flags	0x21

This adds the following two constants into the software:

```
{PIC Constants}
MasterPIC = $20;
MasterOCW1 = $21;
```

After consulting the PIC IRQ table[6] we need to add the following line to the software in order to enable IRQ4 (used for COM1 typically):

```
Port[MasterOCW1] := Port[MasterOCW1] and $EF;
```

When we do the "cleanup" when the program finishes, we also need to disable this IRQ as well with this line of software:

```
Port[MasterOCW1] := Port[MasterOCW1] or $10;
```

Remember that COM2 is on another IRQ vector, so you will have to use different constants for that IRQ. That will be demonstrated a little bit later. We are using a logical and/or with the existing value in this PIC register because we don't want to change the values for the other interrupt vectors that other software and drivers may be using on your PC.

6 http://en.wikibooks.org/wiki/Serial%20Programming%3A8250%20UART%20Programming%23PIC%20Device%20Masking

We also need to modify the Interrupt Service Routine (ISR) a little bit to work with the PIC. There is a command you can send to the PIC that is simply called End of Interrupt (EOI). This signals to the PIC that it can clear this interrupt signal and process lower-priority interrupts. If you fail to clear the PIC, the interrupt signal will remain and none of the other interrupts that are "lower priority" can be processed by the CPU. This is how the CPU communicates back to the PIC to end the interrupt cycle.

The following line is added to the ISR to make this happen:

```
Port[MasterPIC] := EOI;
```

4.7.4 Modem Control Register

This is perhaps the most non-obvious little mistake you can make when trying to get the UART interrupt. The Modem Control register is really the way for the UART to communicate to the rest of the PC. Because of the way the circuitry on the motherboards of most computers is designed, you usually have to turn on the Auxiliary Output 2 signal in order for interrupts to "connect" to the CPU. In addition, here we are going to turn on the RTS and DTS signals on the serial data cable[7] to make sure the equipment is going to transmit. We will cover software and hardware flow control in a later section.

To turn on these values in the MCR, we need to add the following line in the software:

```
Port[ComPort[1] + MCR] := $0B;
```

4.7.5 Interrupt Enable Register

We are still not home free yet. We still need to enable interrupts on the UART itself. This is very simple, and for now all we want to trigger an interrupt from the UART is just when data is received by the UART. This is a very simple line to add here:

```
Port[ComPort[1] + IER] := $01;
```

4.7.6 Putting this together so far

Here is the complete program using ISR input:

```
program ISRTerminal;
uses
  Crt, Dos;
const
  {UART Constants}
  THR = 0;
```

7 http://en.wikibooks.org/wiki/Serial%20Programming%3ARS-232%20Connections

```
  RBR = 0;
  IER = 1;
  FCR = 2;
  LCR = 3;
  MCR = 4;
  LSR = 5;
  Latch_Low = $00;
  Latch_High = $01;
  {PIC Constants}
  MasterPIC = $20;
  MasterOCW1 = $21;
  {Character Constants}
  NullLetter = #0;
  EscapeKey = #27;
var
  ComPort: array [1..4] of Word absolute $0040:$0000;
  OldSerialVector: procedure;
  OutputLetter: Char;
{$F+}
procedure SerialDataIn; interrupt;
var
  InputLetter: Char;
begin
  if (Port[ComPort[1] + LSR] and $01) > 0 then begin
    InputLetter := Chr(Port[ComPort[1] + RBR]);
    Write(InputLetter);
  end; {if}
  Port[MasterPIC] := EOI;
end;
{$F-}
begin
  Writeln('Simple Serial ISR Data Terminal Program.  Press "Esc" to
quit.');
  {Change UART Settings}
  Port[ComPort[1] + LCR] := $80;
  Port[ComPort[1] + Latch_High] := $00;
  Port[ComPort[1] + Latch_Low] := $0C;
  Port[ComPort[1] + LCR] := $03;
  Port[ComPort[1] + FCR] := $07; {clearing the FIFOs}
  Port[ComPort[1] + FCR] := $00; {disabling FIFOs}
  Port[ComPort[1] + MCR] := $0B;
  {Setup ISR vectors}
  GetIntVec($0C,@OldSerialVector);
  SetIntVec($0C,Addr(SerialDataIn));
  Port[MasterOCW1] := Port[MasterOCW1] and $EF;
  Port[ComPort[1] + IER] := $01;
  {Scan for keyboard data}
  OutputLetter := NullLetter;
  repeat
    if KeyPressed then begin
      OutputLetter := ReadKey;
      Port[ComPort[1] + THR] := Ord(OutputLetter);
    end; {if}
  until OutputLetter = EscapeKey;
  {Put the old ISR vector back in}
  SetIntVec($0C,@OldSerialVector);
  Port[MasterOCW1] := Port[MasterOCW1] or $10;
end.
```

At this point you start to grasp how complex serial data programming can get. We are not finished yet, but if you have made it this far you hopefully understand each part of the program listed above. We are going to try and stay with this one step at a time, but at this point you should be able to write some simple custom software that uses serial I/O.

4.7.7 Command Line Input

There are a number of different ways that you can "scan" the parameters that start the program. For example, if you start a simple terminal program in DOS, you can use this command to begin:

```
C:> terminal COM1 9600 8 1 None
```

or perhaps

```
C:> terminal COM4 1200 7 2 Even
```

Obviously there should not be a need to have the end-user recompile the software if they want to change something simple like the baud rate. What we are trying to accomplish here is to grab those other items that were used to start the program. In Turbo Pascal, there is function that returns a string

```
ParamStr(index)
```

which contains each item of the command line. These are passed to the program through strings. A quick sample program on how to extract these parameters can be found here:

```
program ParamTst;
var
  Index: Integer;
begin
  writeln('Parameter Test -- displays all command line parameters of
this program');
  writeln('Parameter Count = ',ParamCount);
  for Index := 0 to ParamCount do begin
    writeln('Param # ',Index,' - ',ParamStr(Index));
  end;
end.
```

One interesting "parameter" is parameter number 0, which is the name of the program that is processing the commands. We will not be using this parameter, but it is something useful in many other programming situations.

4.7.8 Grabbing Terminal Parameters

For the sake of simplicity, we are going to require that either all of the parameters are going to be in that format of baud rate, bit size, stop bits, parity; or there will be no parameters at all. This example is going to be mainly to demonstrate how to use variables to change the settings of the UART by the software user rather than the programmer. Since the added sections are self-explanatory, I'm just going to give you the complete program. There will be some string manipulation going on here that is beyond the scope of this book, but that is going to be used only for parsing the commands. To keep the user interface simple, we are using the command line arguments alone for changing the UART parameters. We could

build a fancy interface to allow these settings to be changed while the program is running, but that is an exercise that is left to the reader.

Category:Serial Programming[8]

8 http://en.wikibooks.org/wiki/Category%3ASerial%20Programming

5 Serial Linux

5.1 The Classic Unix C APIs for Serial Communication

5.1.1 Introduction

Scope

This page talks about the classic Unix C APIs for controlling serial devices. Languages other than C might provide appropriate wrappers to these APIs which look similar, or come with their own abstraction (e.g. Java[1]). Nevertheless, these APIs are the lowest level of abstraction one can find for serial I/O in Unix. And, in fact they are also the highest abstraction in C on standard Unix. Some Unix versions ship additional vendor-specific proprietary high-level APIs. These APIs are not discussed here.

Actual implementations of classic Unix serial APIs do vary in practice, due to the different versions of Unix and its clones, like Linux. Therefore, this module just provides a general outline. It is highly recommended that you study a particular Unix version's manual (man pages) when programming for a serial device in Unix. The relevant man pages are not too great a read, but they are usually complete in their listing of options and parameters. Together with this overview it should be possible to implement programs doing serial I/O under Unix.

Basics

Linux, or any Unix, is a multi-user, multi-tasking operating system. As such, programs usually don't, and are usually not allowed to, access hardware resources like serial UARTs directly. Instead, the operating system provides

1. low-level drivers for mapping the device into the file system (`/dev` and/or `/device/` file system entries),
2. the standard system calls for opening, reading, writing, and closing the device, and
3. the standard system call for controlling a device, and/or
4. high-level C libraries for controlling the device.

The low-level driver not only maps the device into the file system with the help of the kernel, it also encapsulates the particular hardware. The user often does not even know or care what type of UART is in use.

1 Chapter 6 on page 87

Classic Unix systems often provide two different device nodes (or minor numbers) for serial I/O hardware. These provide access to the same physical device via two different names in the `/dev` hierarchy. Which node is used affects how certain serial control signals, such as DCD (data carrier detect), are handled when the device is opened. In some cases this can be changed programmatically, making the difference largely irrelevant. As a consequence, Linux only provides the different devices for legacy programs.

Device names in the file system can vary, even on the same Unix system, as they are simply aliases. The important parts of a device name (such as in `/dev`) are the major and minor numbers. The major number distinguishes a serial port, for example, from a keyboard driver, and is used to select the correct driver in the kernel. Note that the major number differs between different Unix systems. The minor number is interpreted by the device driver itself. For serial device drivers, it is typically used to detect which physical interface to use. Sometimes, the minor number will also be used by the device driver to determine the DCD behavior or the hardware flow control signals to be used.

The typical (but not standardized, see above) device names under Unix for serial interfaces are:

/dev/tty*xxx*

Normal, generic access to the device. Used for terminal and other serial communication (originally for **tele**ty**pes). More recently, they are also used in modem communication, for example, whereas the `/dev/cua`*`xxx`* was used on older systems.

See the following module on how terminal I/O and serial I/O relate on Unix.

/dev/cua*xxx*

Legacy device driver with special DCD handling. Typically this was used for accessing a modem on old Unix systems, such as running the UUCP[2] communication protocol over the serial line and the modem. The *cu* in the name stands for the #`cu`[3] program. The *a* for ACU (automatic call unit).

The **xxx** part in the names above is typically a one or two digit number, or a lowercase letter, starting at 'a' for the first interface.

PC-based Unix systems often mimic the DOS/Windows naming for the devices and call them `/dev/com`*`xxx`*.

To summarize, when programming for the serial interface of a Unix system it is **highly advisable** to provide complete configuration for the device name. Not even the typical `/dev` path should be hard coded.

Note, devices with the name `/dev/pty`*`xxx`* are pseudo terminal devices, typically used by a graphical user interface to provide a terminal emulator like *xterm* or *dtterm* with a "terminal" device, and to provide a terminal device for network logins. There is no serial hardware behind these device drivers.

2 `http://en.wikipedia.org/wiki/UUCP`
3 Chapter 5.2.6 on page 84

5.1.2 Serial I/O via Terminal I/O

Basics

Serial I/O under Unix is implemented as part of the terminal I/O capabilities of Unix. And the terminal I/O capabilities of Unix were originally the typewriter/teletype capabilities. Terminal I/O is not limited to terminals, though. The terminal I/O API is used for communication with many serial devices other than terminals, such as modems and printers.

The terminal API itself has evolved over time. These days three terminal APIs are still used in Unix programs and can be found in recent Unix implementations. A fourth one, the very old one from Unix Version 6 exists, but is quite rare these days.

The three common ones are:

1. V7, 4BSD, XENIX style device-specific ioctl-based API[4],
2. An old one called termio[5]
3. A newer one (although still already a few decades old), which is called termios[6] (note the additional 's').

The newer `termios` API is based on the older `termio` API, and so the two `termio...` APIs share a lot of similarities. The `termios` API has also undergone changes since inception. For example, the method of specifying the baud rate has changed from using pre-defined constants to a more relaxed schema (the constants can still be used as well on most implementations).

Systems that support the newer `termios` often also support the older `termio` API, either by providing it in addition, or by providing a `termios` implementation with data structures which can be used in place of the `termio` data structures and work as `termio`. These systems also often just provide one man page under the older name **termio***(7)* which is then in fact the `termios` man page, too.

In addition, some systems provide other, similar APIs, either in addition or as a replacement. `termiox` is such an API, which is largely compatible with `termio` and adds some extensions to it taken from `termios`. So `termiox` can logically be seen as an intermediate step between `termio` and `termios`.

The terminal I/O APIs rely on the standard system calls for reading and writing data. They don't provide their own reading/writing functions. Reading and writing data is done via the **read***(2)* and **write***(2)* system calls. The terminal I/O APIs just add functions for controlling and configuring the device. Most of this happens via the **ioctl***(2)* system call.

Unfortunately, whichever of the standard APIs is used, one fact holds for all of them: They are a slight mess. Well, not really. Communication with terminals was and is a difficult issue, and the APIs reflect these difficulties. But due to the fact that one can do "everything" with the APIs, it is overwhelming when one "just" wants to do some serial communication. So why is there no separate serial-I/O-only API in Unix? There are probably two reasons for this:

---

4 `http://en.wikibooks.org/wiki/Serial_Programming%3ASerial_Linux%23V7%20%2F%20ioctl%282%29`
5 `http://en.wikibooks.org/wiki/Serial_Programming%3ASerial_Linux%23termio%20%2F%20ioctl%282%29`
6 `http://en.wikibooks.org/wiki/Serial_Programming%3ASerial_Linux%23termios`

1. Terminals/teletypes were the first, and apparently very important, serial devices which were connected to Unix. So that API was created first.
2. Once the API was there, there was no need to create a separate one for serial I/O only, since a large part of terminal I/O is serial I/O, and all needed features were already there in the terminal I/O API.

So which API should one use? There is one good reason to use the old V7 API. It is the simplest among the APIs - after going through some initialization woes on modern Unix systems. In general, however, the newer `termios` API makes the most sense, although it is the most complex one.

Line Discipline

When programming serial interfaces on Unix, there is one phrase - *line discipline* - which can drive programmers crazy. The line discipline provides the hardware-independent interface for the communication between the computer and the terminal device. It handles such things as editing, job control, and special character interpretation, and performs transformations on the incoming and outgoing data.

This is useful for terminal communication (e.g. when a backspace character should erase the latest character from the send buffer before it goes over the wire, or when different end-of-line character sequences between the terminal and the computer need to be converted). These features are, however, hardly useful when communicating with the plethora of other serial devices, where unaltered data communication is desired.

Much of the serial programming in Unix is hitting the line discipline which is in use over the head so it doesn't touch the data. Monitoring what actually goes over the wire is a good idea.

5.1.3 Unix V6/PWB

Unix *Bell Version 6* with the *programmer's workbench* (PWB) was released in 1975 to universities. It was the first Unix with an audience outside AT&T. It already had a terminal programming API. Actually, at that point it was the *typewriter* API. That API is not described here in depth.

The usage of this API can in theory be identified by the presence of the following signature in some source code:

```
#include <sgtty.h>
stty(fd, data)
int fd;
char *data;

gtty(fd, data)
int fd;
char *data;
```

In theory, because at that time the C language was still a little bit different.

`data` is supposed to point to a

```
struct {
   char ispeed, ospeed;
   char erase, kill;
   int  mode;
} *data;
```

structure. That structure later became `struct sgttyb` in Unix V7. Finding the V6 API in source code should be rare. Anyhow, recent Unix versions and clones typically don't support this API any more.

5.1.4 Unix V7

See Serial Programming:Unix/V7[7]

5.1.5 termios

A simple terminal program with termios.h can look like this:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>

int main(int argc,char** argv)
{
        struct termios tio;
        struct termios stdio;
        struct termios old_stdio;
        int tty_fd;

        unsigned char c='D';
        tcgetattr(STDOUT_FILENO,&old_stdio);

        printf("Please start with %s /dev/ttyS1 (for
 example)\n",argv[0]);
        memset(&stdio,0,sizeof(stdio));
        stdio.c_iflag=0;
        stdio.c_oflag=0;
        stdio.c_cflag=0;
        stdio.c_lflag=0;
        stdio.c_cc[VMIN]=1;
        stdio.c_cc[VTIME]=0;
        tcsetattr(STDOUT_FILENO,TCSANOW,&stdio);
        tcsetattr(STDOUT_FILENO,TCSAFLUSH,&stdio);
        fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);       // make the
 reads non-blocking

        memset(&tio,0,sizeof(tio));
        tio.c_iflag=0;
        tio.c_oflag=0;
        tio.c_cflag=CS8|CREAD|CLOCAL;           // 8n1, see termios.h
```

7 http://en.wikibooks.org/wiki/Serial%20Programming%3AUnix%2FV7

```
 for more information
        tio.c_lflag=0;
        tio.c_cc[VMIN]=1;
        tio.c_cc[VTIME]=5;

        tty_fd=open(argv[1], O_RDWR | O_NONBLOCK);
        cfsetospeed(&tio,B115200);            // 115200 baud
        cfsetispeed(&tio,B115200);            // 115200 baud

        tcsetattr(tty_fd,TCSANOW,&tio);
        while (c!=,q,)
        {
                if (read(tty_fd,&c,1)>0)
 write(STDOUT_FILENO,&c,1);              // if new data is available
 on the serial port, print it out
                if (read(STDIN_FILENO,&c,1)>0)  write(tty_fd,&c,1);
                   // if new data is available on the console, send it
 to the serial port
        }

        close(tty_fd);
        tcsetattr(STDOUT_FILENO,TCSANOW,&old_stdio);

        return EXIT_SUCCESS;
}
```

See Serial_Programming:Unix/termios[8]

5.1.6 termio / ioctl(2)

See Serial Programming:Unix/termio[9]

5.2 Serial I/O on the Shell Command Line

5.2.1 Introduction

It is possible to do serial I/O on the Unix command line. However, the available control is limited. Reading and writing data can be done with the shell I/O redirections like <, >, and |. Setting basic configuration, like the baud rate, can be done with the `stty` (set terminal type) command.

There is also `libserial` for Linux. It's a simple C++ class which hides some of the complexity of termios.

5.2.2 Configuration with `stty`

The Unix command `stty` allows one to configure a "terminal". Since all serial I/O under Unix is done via terminal I/O, it should be no surprise that stty can also be used to configure serial lines. Indeed, the options and parameters which can be set via stty often have a 1:1

8 http://en.wikibooks.org/wiki/Serial_Programming%3AUnix%2Ftermios
9 http://en.wikibooks.org/wiki/Serial%20Programming%3AUnix%2Ftermio

mapping to termio/termios. If the explanations regarding an option in the **stty***(1)* man page is not sufficient, looking up the option in the termio/termios man page can often help.

On "modern" (System V) Unix versions, stty changes the parameters of its current **standard input**. On older systems, stty changes the parameters of its current **standard output**. We assume a modern Unix is in use here. So, to change the settings of a particular serial interface, its device name must be provided to stty via an I/O redirect:

```
stty parameters < /dev/com0  # change setting of /dev/com0
```

On some systems, the settings done by stty are reverted to system defaults as soon as the device is closed again. This closing is done by the shell as soon as the `stty parameters < /dev/com0` command has finished. So when using the above command, the changes will only be in effect for a few milliseconds.

One way to keep the device open for the duration of the communication is to start the whole communication in a sub shell (using, for example, '(...)'), and redirecting that input. So to send the string "ATI0" over the serial line, one could use:

```
( stty parameters
  echo "ATI0"
) < /dev/com0 > /dev/com0
```

Interweaving sending and receiving data is difficult from the command line. Two processes are needed; one reading from the device, and the other writing to the device. This makes it difficult to coordinate commands sent with the responses received. Some extensive shell scripting might be needed to manage this.

A common way to organize the two processes is to put the reading process in the background, and let the writing process continue to run in the foreground. For example, the following script configures the device and starts a background process for copying all received data from the serial device to standard output. Then it starts writing commands to the device:

```
# Set up device and read from it.
# Capture PID of background process so it is possible
# to terminate background process once writing is done
# TODO: Also set up a trap in case script is killed
#       or crashes.
( stty parameters; cat; )& < /dev/com0
bgPid=$?

# Read commands from user, send them to device
while read cmd; do
   echo "$cmd"
done >/dev/com0

# Terminate background read process
kill $bgPid
```

If there is a chance that a response to some command might never come, and if there is no other way to terminate the process, it is advisable to set up a timeout by using the alarm

signal and `trap` that signal (signal 14), or simply kill the process:

```
trap timeout 14
timeout() {
   echo "timeout occurred"
}
pid=$$
( sleep 60 ; kill -14 $pid; )& # send alarm signal after 60 sec.
# normal script contents goes here
```

or

```
pid=$$
( sleep 60; kill -9 $pid;)& # brutally kill process after 60 sec.
# normal script contents goes here
```

5.2.3 Permanent Configuration

Overview

It is possible to provide a serial line with a default configuration. On classic Unix this is done with entries in the `/etc/ttytab` configuration file, on newer (System V R4) systems with `/etc/ttydefs`.

The default configurations make some sense when they are used for setting up terminal lines or dialup lines for a Unix system (and that's what they are for). However, such default configurations are not of much use when doing some serial communication with some other device. The correct function of the communication program should better not depend on some operating system configuration. Instead, the application should be self-contained and configure the device as needed by it.

`/etc/ttytab`

The ttytab format varies from Unix to Unix, so checking the corresponding man page is a good idea. If the device is not intended for a terminal (no login), then the *getty* field (sometimes also called the program field, usually the 3rd field) for the device entry should be empty. The init field (often the 4th field) can contain an initialization command. Using `stty` here is a good idea. So, a typical entry for a serial line might look like:

```
# Device    TermType    Getty   Init
tty0        unknown     ""      "stty parameters"
```

`/etc/ttydefs`

Just some hints:

`/etc/ttydefs` provides the configuration as used by the **ttymon** program. The settings are similar to the settings possible with stty.

`ttymon` is a program which is typically run under control of the Service Access Controller (SAC), as part of the Service Access Facility (SAF).

TODO: Provide info to set up all the sac/sacadm junk.

/etc/serial.conf

Just some hints:

A Linux-specific way of configuring serial devices using the **setserial** program.

5.2.4 tty

tty with the **-s** option can be used to test if a device is a terminal (supports the termio/termios ioctl()'s). Therefore it can also be used to check if a given file name is indeed a device name of a serial line.

```
echo "Enter serial device name: \c"
read dev
if tty -s < "$dev"; then
  echo "$dev is indeed a serial device."
else
  echo "$dev is not a serial device."
fi
```

5.2.5 tip

It is a simple program for establishing a terminal connection with a remote system over a serial line. `tip` takes the necessary communication parameters, including the parameters for the serial communication, from a tip-specific configuration file. Details can be found in the `tip`*(1)* manual page.

Example:

To start the session over the first serial interface (here `ttya`):

```
tip -9600 /dev/ttya
```

To leave the session:

```
~.
```

5.2.6 uucp

Overview

Uucp (Unix-to-Unix-Copy) is a set of programs for moving data over serial lines/modems between Unix computers. Before the rise of the Internet uucp was the heart and foundation of services like e-mail and Usenet (net news) between Unix computers. Today uucp is largely insignificant. However, it is still a good choice if two or more Unix systems should be connected via serial lines/modems.

The uucp suite also contains command line tools for login over a serial line (or another UUCP bearer to a remote system. These tools are `cu` and `ct`. They are e.g. useful when trying to access a device connected via a serial line and when debugging some serial line protocol.

cu

`cu` "call another UNIX system", does what the name implies. Only, that the other system does not have to be a UNIX system at all. It just sets up a serial connection, possibly by dialing via a modem.

`cu` is the oldest Unix program for serial communication. It's the reason why some serial devices on classic Unix systems are called something like `/dev/cul0` and `/dev/cua0`. Where *cu* of course stands for the `cu` program supposed to use the devices, *l* stands for *line* - the communication line, and *a* for acu (automatic call unit).

Note:
An ACU is kind of a modem. Modern modems work slightly different and don't provide separate serial interfaces for dialing and communicating with the remote side. Instead they do both over the same serial interface, using some kind of inband signaling. See Serial Programming:Modems and AT Commands[a].

[a] http://en.wikibooks.org/wiki/Serial%20Programming%3AModems%20and%20AT%20Commands

ct

`ct` is intended to spawn a login to a remote system over a modem line, serial line, or similar bearer. It uses the uucp devices list to find the necessary dialing (modem) commands, and the serial line settings.

5.3 System Configuration

inittab, ttytab, SAF configuration

5.4 Other Serial Programming Articles

Category:Serial Programming[10]

10 http://en.wikibooks.org/wiki/Category%3ASerial%20Programming

6 Serial Java

6.1 Using Java for Serial Communication

6.1.1 Introduction

Because of Java's platform-independence, serial interfacing is difficult. Serial interfacing requires a standardized API with platform-specific implementations, which is difficult for Java.

Unfortunately, Sun doesn't pay much attention to serial communication in Java. Sun has defined a serial communication API, called *JavaComm*[1], but an implementation of the API is not part of the Java standard edition. Sun provides a reference implementation for a few, but not all Java platforms. Particularly, at the end of 2005 Sun silently withdrew *JavaComm* support for Windows. Third party implementations for some of the omitted platforms are available. *JavaComm* hasn't seen much in the way of maintenance activities, only the bare minimum maintenance is performed by Sun, except that Sun has apparently responded to pressure from buyers of their own Sun Ray thin clients and has adapted JavaComm to this platform while dropping Windows support.

This situation, and the fact that Sun originally did not provide a *JavaComm* implementation for Linux (starting in 2006, they now do) led to the development of the free-software *RxTx*[2] library. *RxTx* is available for a number of platforms, not only Linux. It can be used in conjunction with *JavaComm* (*RxTx* providing the hardware-specific drivers), or it can be used stand-alone. When used as a *JavaComm* driver the bridging between the *JavaComm* API and *RxTx* is done by *JCL* (*JavaComm for Linux*). *JCL* is part of the *RxTx* distribution.

Sun's negligence of *JavaComm* and *JavaComm****s particular programming model gained*** **JavaComm*****the reputation of being unusable. Fortunately, this is not the case. Unfortunately, the reputation is further spread by people who don't know the basics of serial programming at all and make*** **JavaComm*****responsible for their lack of understanding.***

RxTx - if not used as a *JavaComm* driver - provides a richer interface, but one which is not standardized. *RxTx* supports more platforms than the existing *JavaComm* implementations. Recently, *RxTx* has been adopted to provide the same interface as *JavaComm*, only that the package names don't match Sun's package names.

So, which of the libraries should one use in an application? If maximum portability (for some value of "maximum") is desired, then *JavaComm* is a good choice. If there is no *JavaComm* implementation for a particular platform available, but an *RxTx* implementation

1 `http://www.oracle.com/technetwork/java/index-jsp-141752.html`
2 `http://rxtx.qbang.org/wiki/index.php/Main_Page`

is, then *RxTx* could be used as a driver on that platform for *JavaComm*. So, by using JavaComm one can support all platforms which are either directly supported by Sun's reference implementation or by RxTx with JCL. This way the application doesn't need to be changed, and can work against just one interface, the standardized *JavaComm* interface.

This module discusses both *JavaComm* and *RxTx*. It mainly focuses on demonstrating concepts, not ready-to-run code. Those who want to blindly copy code are referred to the sample code that comes with the packages. Those who want to know what they are doing might find some useful information in this module.

6.1.2 Getting started

- Learn the basics of serial communication and programming[3].
- Have the documentation of the device you want to communicate with (e.g. the modem) ready.
- Set up all hardware and a test environment
- Use, for example, a terminal program to manually communicate with the device. This is to be sure the test environment is set up correctly and you have understood the commands and responses from the device.
- Download the API implementation you want to use for your particular operating system

- Read
  - the JavaComm and/or RxTx installation instruction (and follow it)
  - the API documentation
  - the example source code shipped

6.1.3 Installation

General Issues

Both *JavaComm* and *RxTX* show some installation quirks. It is highly recommended to follow the installation instructions word-for-word. If they say that a jar file or a shared library has to go into a particular directory, then this is meant seriously! If the instructions say that a particular file or device needs to have a specific ownership or access rights, this is also meant seriously. Many installation troubles simply come from not following the instructions precisely.

It should especially be noted that some versions of *JavaComm* come with two installation instructions. One for Java 1.2 and newer, one for Java 1.1. Using the wrong one will result in a non-working installation. On the other hand, some versions/builds/packages of *RxTx* come with incomplete instructions. In such a case the corresponding source code distribution of *RxTx* needs to be obtained, which should contain complete instructions.

It should be further noticed that it is also typical for Windows JDK installations to come with up to three VMs, and thus three extension directories.

- One as part of the JDK,

3 http://en.wikibooks.org/wiki/Serial%20Programming

- one as part of the private JRE which comes with the JDK to run JDK tools, and
- one as part of the public JRE which comes with the JDK to run applications

Some even claim to have a fourth JRE somewhere in the \Windows directory hierarchy.

JavaComm should at least be installed as extension in the JDK and in all public JREs.

Webstart

JavaComm

A general problem, both for *JavaComm* and *RxTx* is, that they resist installation via Java WebStart[4]:

JavaComm is notorious, because it requires a file called *javax.comm.properties* to be placed in the JDK lib directory, something which can't be done with Java WebStart. This is particularly sad, because the need for that file is the result of some unnecessary design/decision in *JavaComm* and could have easily been avoided by the *JavaComm* designers. Sun constantly refuses to correct this error, citing the mechanism is essential. Which is, they are lying through their teeth when it comes to *JavaComm*, particular, because Java for a long time has a service provider architecture exactly intended for such purposes.

The contents of the properties file is typically just one line, the name of the java class with the native driver, e.g.:

```
driver=com.sun.comm.Win32Driver
```

The following is a hack which allows to deploy JavaComm via Web Start ignoring that brain-dead properties file. It has serious drawbacks, and might fail with newer JavaComm releases - should Sun ever come around and make a new version.

First, turn off the security manager. Some doofus programmer at Sun decided that it would be cool to again and again check for the existence of the dreaded *javax.comm.properties* file, even after it has been loaded initially, for no other apparent reason than checking for the file.

```
System.setSecurityManager(null);
```

Then, when initializing the JavaComm API, initialize the driver manually:

```
String driverName = "com.sun.comm.Win32Driver"; // or get as a JNLP
property
CommDriver commDriver =
(CommDriver)Class.forName(driverName).newInstance();
commDriver.initialize();
```

RxTx

4 http://java.sun.com/products/javawebstart/

RxTx on some platforms requires changing ownership and access rights of serial devices. This is also something which can't be done via WebStart.

At startup of your program you could ask the user to perform the necessary setup as super user.

Further, RxTx has a pattern matching algorithm for identifying "valid" serial device names. This often breaks things when one wants to use non-standard devices, like USB-to-serial converters. This mechanism can be overridden by system properties. See the RxTx installation instruction for details.

6.2 JavaComm API

6.2.1 Introduction

The official API for serial communication in Java is the JavaComm API. This API is not part of the standard Java 2 version. Instead, an implementation of the API has to be downloaded separately. Unfortunately, JavaComm has not received much attention from Sun, and hasn't been really maintained for a long time. From time to time Sun does trivial bug-fixes, but doesn't do the long overdue main overhaul.

This section explains the basic operation of the JavaComm API. The provided source code is kept simple to demonstrate important point. It needs to be enhanced when used in a real application.

The source code in this chapter is not the only available example code. The JavaComm download comes with several examples. These examples almost contain more information about using the API than the API documentation. Unfortunately, Sun does not provide any real tutorial or some introductory text. Therefore, it is worth studying the example code to understand the mechanisms of the API. Still, the API documentation should be studied, too. But the best way is to study the examples and play with them. Due to the lack of easy-to-use application and people's difficulty in understanding the APIs programming model, the API is often bad-mouthed. The API is better than its reputation, and functional. But no more.

The API uses a callback mechanism to inform the programmer about newly arriving data. It is also a good idea to study this mechanism instead of relying on polling the port. Unlike other callback interfaces in Java (e.g. in the GUI), this one only allows one listener listening to events. If multiple listeners require to listen to serial events, the one primary listener has to be implemented in a way that it dispatches the information to other secondary listeners.

6.2.2 Download & Installation

Download

Sun's JavaComm[5] web page points to a download location[6]. Under this location Sun currently (2007) provides JavaComm 3.0 implementations for Solaris/SPARC, Solaris/x86, and Linux x86. Downloading requires to have registered for a Sun Online Account. The download page provides a link to the registration page. The purpose of this registration is unclear. One can download JDKs and JREs without registration, but for the almost trivial JavaComm Sun cites legal and governmental restrictions on the distribution and exportation of software.

The Windows version of JavaComm is no longer officially available, and Sun has - against their own product end-of-live policy - not made it available in the Java products archive[7]. However, the 2.0 Windows version (javacom 2.0) is still downloadable from here[8].

Installation

Follow the installation instructions that come with the download. Some versions of Java-Comm 2.0 come with two installation instructions. The most obvious of the two instructions is unfortunately the wrong one, intended for ancient Java 1.1 environments. The information referring to the also ancient Java 1.2 (jdk1.2.html) is the right one.

Particularly Windows users are typically not aware that they have copies of the same VM installed in several locations (typically three to four). Some IDEs also like to come with own, private JRE/JDK installations, as do some Java applications. The installation needs to be repeated for every VM installation (JDKs and JREs) which should be used in conjunction with the development and execution of a serial application.

IDEs typically have IDE-specific ways of how a new library (classes and documentation) is made known to the IDE. Often a library like JavaComm not only needs to be made known to the IDE as such, but also to each project that is supposed to use the library. Read the IDE's documentation. It should be noted that the old JavaComm 2.0 version comes with JavaDoc API documentation that is structured in the historic Java 1.0 JavaDoc layout. Some modern IDEs are no longer aware of this structure and can't integrate the JavaComm 2.0 documentation into their help system. In such a case an external browser is needed to read the documentation (a recommended activity ...).

Once the software is installed it is recommended to examine the samples and JavaDoc directories. It makes sense to build and run one of the sample applications to verify that the installation is correct. The sample applications typically need some minor adaptations in order to run on a particular platform (e.g. changes to the hard-coded com port identifiers). It is a good idea to have some serial hardware, like cabling, a null modem, a breakout box,

5 `http://java.sun.com/products/javacomm/`
6 `http://www.sun.com/download/products.xml?id=43208d3d`
7 `http://java.sun.com/products/archive/`
8 `http://wind.lcs.mit.edu/download/`

a real modem, PABX and others available when trying out a sample application. Serial_-Programming:RS-232 Connections[9] and Serial_Programming:Modems and AT Commands[10] provide some information on how to set up the hardware part of a serial application development environment.

Finding the desired serial Port

The first three things to do when programming serial lines with JavaComm are typically

1. to enumerate all serial ports (port identifiers) available to JavaComm,
2. to select the desired port identifier from the available ones, and
3. to acquire the port via the port identifier.

Enumerating and selecting the desired port identifier is typically done in one loop:

```
import javax.comm.*;
import java.util.*;
...

//
// Platform specific port name, here a Unix name
//
// NOTE: On at least one Unix JavaComm implementation JavaComm
//       enumerates the ports as "COM1" ... "COMx", too, and not
//       by their Unix device names "/dev/tty...".
//       Yet another good reason to not hard-code the wanted
//       port, but instead make it user configurable.
//
String wantedPortName = "/dev/ttya";

//
// Get an enumeration of all ports known to JavaComm
//
Enumeration portIdentifiers =
CommPortIdentifier.getPortIdentifiers();

//
// Check each port identifier if
//   (a) it indicates a serial (not a parallel) port, and
//   (b) matches the desired name.
//
CommPortIdentifier portId = null;  // will be set if port found
while (portIdentifiers.hasMoreElements())
{
    CommPortIdentifier pid = (CommPortIdentifier)
portIdentifiers.nextElement();
    if(pid.getPortType() == CommPortIdentifier.PORT_SERIAL &&
       pid.getName().equals(wantedPortName))
    {
        portId = pid;
        break;
    }
}
if(portId == null)
{
    System.err.println("Could not find serial port " +
```

9 http://en.wikibooks.org/wiki/Serial_Programming%3ARS-232%20Connections
10 http://en.wikibooks.org/wiki/Serial_Programming%3AModems%20and%20AT%20Commands

```
wantedPortName);
    System.exit(1);
}

//
// Use port identifier for acquiring the port
//
...
```

Note:
JavaComm itself obtains the default list of available serial port identifiers from its platform-specific driver. The list is not really configurable via JavaComm. The method `CommPortIdentifier.addPortName()` is misleading, since driver classes are platform specific and their implementations are not part of the public API. Depending on the driver, the list of ports might be configurable / expendable in the driver. So if a particular port is not found in JavaComm, sometimes some fiddling with the driver can help.

Once a port identifier has been found, it can be used to acquire the desired port:

```
//
// Use port identifier for acquiring the port
//
SerialPort port = null;
try {
    port = (SerialPort) portId.open(
        "name", // Name of the application asking for the port
        10000   // Wait max. 10 sec. to acquire port
    );
} catch(PortInUseException e) {
    System.err.println("Port already in use: " + e);
    System.exit(1);
}
//
// Now we are granted exclusive access to the particular serial
// port. We can configure it and obtain input and output streams.
//
...
```

6.2.3 Initialize a Serial Port

The initialization of a serial port is straight forward. Either individually set the communication preferences (baud rate, data bits, stop bits, parity) or set them all at once using the setSerialPortParams(...) convenience method.

As part of the initialization process the Input and Output streams for communication will be configured in the example.

```
import java.io.*;
...


//
// Set all the params.
```

```
// This may need to go in a try/catch block which throws
UnsupportedCommOperationException
//
port.setSerialPortParams(
    115200,
    SerialPort.DATABITS_8,
    SerialPort.STOPBITS_1,
    SerialPort.PARITY_NONE);


//
// Open the input Reader and output stream. The choice of a
// Reader and Stream are arbitrary and need to be adapted to
// the actual application. Typically one would use Streams in
// both directions, since they allow for binary data transfer,
// not only character data transfer.
//
BufferedReader is = null;  // for demo purposes only. A stream would
be more typical.
PrintStream    os = null;


try {
  is = new BufferedReader(new
InputStreamReader(port.getInputStream()));
} catch (IOException e) {
  System.err.println("Can't open input stream: write-only");
  is = null;
}


//
// New Linux systems rely on Unicode, so it might be necessary to
// specify the encoding scheme to be used. Typically this should
// be US-ASCII (7 bit communication), or ISO Latin 1 (8 bit
// communication), as there is likely no modem out there accepting
// Unicode for its commands. An example to specify the encoding
// would look like:
//
//     os = new PrintStream(port.getOutputStream(), true,
"ISO-8859-1");
//
os = new PrintStream(port.getOutputStream(), true);


//
// Actual data communication would happen here
// performReadWriteCode();
//


//
// It is very important to close input and output streams as well
// as the port. Otherwise Java, driver and OS resources are not
released.
//
if (is != null) is.close();
if (os != null) os.close();
if (port != null) port.close();
```

6.2.4 Simple Data Transfer

Simple Writing of Data

Writing to a serial port is as simple as basic Java IO. However there are a couple of caveats to look out for if you are using the AT Hayes protocol:

1. Don't use println (or other methods that automatically append "\n") on the OutputStream. The AT Hayes protocol for modems expects a "\r\n" as the delimiter (regardless of underlying operating system).
2. After writing to the OutputStream, the InputStream buffer will contain a repeat of the command that was sent to it (with line feed), if the modem is set to echoing the command line, and another line feed (the answer to the "AT" command). So as part of the write operation make sure to clean the InputStream of this information (which can actually be used for error detection).
3. When using a Reader/Writer (not a really good idea), at least set the character encoding to US-ASCII instead of using the platform's default encoding, which might or might not work.
4. Since the main operation when using a modem is to transfer data unaltered, the communication with the modem should be handled via InputStream/OutputStream, and not a Reader/Writer.

```
// Write to the output
os.print("AT");
os.print("\r\n"); // Append a carriage return with a line feed

is.readLine(); // First read will contain the echoed command you
sent to it. In this case: "AT"
is.readLine(); // Second read will remove the extra line feed that
AT generates as output
```

Simple Reading of Data (Polling)

If you correctly carried out the write operation (see above) then the read operation is as simple as one command:

```
// Read the response
String response = is.readLine(); // if you sent "AT" then response
== "OK"
```

Problems with the simple Reading / Writing

The simple way of reading and/or writing from/to a serial port as demonstrated in the previous sections has serious drawbacks. Both activities are done with *blocking I/O*. That means, when there is

- no data available for reading, or
- the output buffer for writing is full (the device does not accept (any more) data),

the read or write method (`os.print()` or `is.readLine()` in the previous example) do not return, and the application comes to a halt. More precisely, the thread from which the read or write is done gets blocked. If that thread is the main application thread, the application freezes until the blocking condition is resolved (data becomes available for reading or device accepts data again).

Unless the application is a very primitive one, freezing of the application is not acceptable. For example, as a minimum some user interaction to cancel the communication should still be possible. What is needed is *non-blocking I/O* or *asynchronous I/O*. However, JavaComm is based on Java's standard blocking I/O system (`InputStream`, `OutputStream`), but with a twist, as shown later.

The mentioned "twist" is that JavaComm provides some limited support for *asynchronous I/O* via an event notification mechanism. But the general solution in Java to achieve *non-blocking I/O* on top of the blocking I/O system is to use threads. Indeed, this is a viable solution for serial writing, and it is strongly recommended to use a separate thread to write to the serial port - even if the event notification mechanism is used, as explained later.

Reading could also be handled in a separate thread. However, this is not strictly necessary if the JavaComm event notification mechanism is used. So summarize:

Activity	**Architecture**
reading	use event notification and/or separate thread
writing	always use separate thread, optionally use event notification

The following sections provide some details.

6.2.5 Event Driven Serial Communication

Introduction

The JavaComm API provides an event notification mechanism to overcome the problems with *blocking I/O*. However, in the typical Sun manner this mechanism is not without problems.

In principle an application can register event listeners with a particular `SerialPort` to be kept informed about important events happening on that port. The two most interesting event types for reading and writing data are

- `javax.comm.SerialPortEvent.DATA_AVAILABLE` and
- `javax.comm.SerialPortEvent.OUTPUT_BUFFER_EMPTY`.

But there are also two problems:

1. Only one single event listener per `SerialPort` can be registered. This forces the programmer to write "monster" listeners, discriminating according to the event type.
2. `OUTPUT_BUFFER_EMPTY` is an optional event type. Well hidden in the documentation Sun states that not all JavaComm implementations support generating events of this type.

Before going into details, the next section will present the principal way of implementing and registering a serial event handler. Remember, there can only be one handler at all, and it will have to handle all possible events.

Setting up a serial Event Handler

```
import javax.comm.*;


/**
 * Listener to handle all serial port events.
 *
 * NOTE: It is typical that the SerialPortEventListener is
implemented
 *       in the main class that is supposed to communicate with the
 *       device. That way the listener has easy access to state
information
 *       about the communication, e.g. when a particular
communication
 *       protocol needs to be followed.
 *
 *       However, for demonstration purposes this example implements
a
 *       separate class.
 */
class SerialListener implements SerialPortEventListener {


    /**
     * Handle serial events. Dispatches the event to event-specific
     * methods.
     * @param event The serial event
     */
    @Override
    public void serialEvent(SerialPortEvent event){


        //
        // Dispatch event to individual methods. This keeps this
ugly
        // switch/case statement as short as possible.
        //
        switch(event.getEventType()) {
            case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
                outputBufferEmpty(event);
                break;


            case SerialPortEvent.DATA_AVAILABLE:
                dataAvailable(event);
                break;


/* Other events, not implemented here ->
            case SerialPortEvent.BI:
                breakInterrupt(event);
                break;


            case SerialPortEvent.CD:
                carrierDetect(event);
                break;
```

```
            case SerialPortEvent.CTS:
                clearToSend(event);
                break;


            case SerialPortEvent.DSR:
                dataSetReady(event);
                break;


            case SerialPortEvent.FE:
                framingError(event);
                break;


            case SerialPortEvent.OE:
                overrunError(event);
                break;


            case SerialPortEvent.PE:
                parityError(event);
                break;
            case SerialPortEvent.RI:
                ringIndicator(event);
                break;
<- other events, not implemented here */


        }
    }


    /**
     * Handle output buffer empty events.
     * NOTE: The reception of this event is optional and not
     *       guaranteed by the API specification.
     * @param event The output buffer empty event
     */
    protected void outputBufferEmpty(SerialPortEvent event) {
        // Implement writing more data here
    }


    /**
     * Handle data available events.
     *
     * @param event The data available event
     */
    protected void dataAvailable(SerialPortEvent event) {
        // implement reading from the serial port here
    }
}
```

Once the listener is implemented, it can be used to listen to particular serial port events. To do so, an instance of the listener needs to be added to the serial port. Further, the reception of each event type needs to be requested individually.

```
SerialPort port = ...;
...
//
// Configure port parameters here. Only after the port is configured
it
```

```
// makes sense to enable events. The event handler might be called
immediately
// after an event is enabled.
...


//
// Typically, if the current class implements the
SerialEventListener interface
// one would call
//
//       port.addEventListener(this);
//
// but for our example a new instance of SerialListener is created:
//
port.addEventListener(new SerialListener());


//
// Enable the events we are interested in
//
port.notifyOnDataAvailable(true);
port.notifyOnOutputEmpty(true);


/* other events not used in this example ->
port.notifyOnBreakInterrupt(true);
port.notifyOnCarrierDetect(true);
port.notifyOnCTS(true);
port.notifyOnDSR(true);
port.notifyOnFramingError(true);
port.notifyOnOverrunError(true);
port.notifyOnParityError(true);
port.notifyOnRingIndicator(true);
<- other events not used in this example */
```

Writing of Data

Setting up a separate Thread for Writing

Using a separate thread for writing has one purpose: Avoiding that the whole application blocks in case the serial port is not ready for writing.

A simple, thread-safe Ring Buffer Implementation

Using a separate thread for writing, separate from some main application thread, implies that there is some way to hand off the data which needs to be written from the application thread to the writing thread. A shared, synchronized data buffer, for example a `byte[]` should do. Further, there needs to be a way for the main application to determine if it can write to the data buffer, or if the data buffer is currently full. In case the data buffer is full it could indicate that the serial port is not ready, and output data has queued up. The main application will have to poll the availability of new space in the shared data buffer. However, between the polling the main application can do other things, for example updating a GUI, providing a command prompt with the ability to abort the sending, etc.

At first glance a `PipedInputStream/PipedOutputStream` pair seems like a good idea for this kind of communication. But Sun wouldn't be Sun if the a piped stream would actually be useful. `PipedInputStream` blocks if the corresponding `PipedOutputStream` is not cleared fast enough. So the application thread would block. Exactly what one wants to avoid by using the separate thread. A `java.nio.Pipe` suffers from the same problem. Its blocking behavior is platform dependent. And adapting the classic I/O used by JavaComm to NIO is anyhow not a nice task.

In this article a very simple synchronized ring buffer is used to hand over the data from one thread to another. In a real world application it is likely that the implementation should be more sophisticated. E.g. in a real world implementation it would make sense to implement OutputStream and InputStream views on the buffer.

A ring buffer as such is nothing special, and has no special properties regarding threading. It is just that this simple data structure is used here to provide data buffering. The implementation is done so that access to this data structure has been made thread safe.

```
/**
 * Synchronized ring buffer.
 * Suitable to hand over data from one thread to another.
 **/
public synchronized class RingBuffer {


    /** internal buffer to hold the data **/
    protected byte buffer[];


    /** size of the buffer **/
    protected int size;


    /** current start of data area **/
    protected int start;


    /** current end of data area **/
    protected int end;




    /**
     * Construct a RingBuffer with a default buffer size of 1k.
     */
    public RingBuffer() {
         this(1024);
    }


    /**
     * Construct a RingBuffer with a certain buffer size.
     * @param size   Buffer size in bytes
     */
    public RingBuffer(int size) {
         this.size = size;
         buffer = new byte[size];
         clear();
    }
```

```
    /**
     * Clear the buffer contents. All data still in the buffer is
lost.
     */
    public void clear() {
        // Just reset the pointers. The remaining data fragments, if
any,
        // will be overwritten during normal operation.
        start = end = 0;
    }


    /**
     * Return used space in buffer. This is the size of the
     * data currently in the buffer.
     * <p>
     * Note: While the value is correct upon returning, it
     * is not necessarily valid when data is read from the
     * buffer or written to the buffer. Another thread might
     * have filled the buffer or emptied it in the mean time.
     *
     * @return currently amount of data available in buffer
     */
    public int data() {
         return start <= end
                    ? end - start
                    : end - start + size;
    }


    /**
     * Return unused space in buffer. Note: While the value is
     * correct upon returning, it is not necessarily valid when
     * data is written to the buffer or read from the buffer.
     * Another thread might have filled the buffer or emptied
     * it in the mean time.
     *
     * @return currently available free space
     */
    public int free() {
         return start <= end
                    ? size + start - end
                    : start - end;
    }


    /**
     * Write as much data as possible to the buffer.
     * @param data   Data to be written
     * @return       Amount of data actually written
     */
    int write(byte data[]) {
        return write(data, 0, data.length);
    }


    /**
     * Write as much data as possible to the buffer.
     * @param data   Array holding data to be written
     * @param off    Offset of data in array
     * @param n      Amount of data to write, starting from off.
     * @return       Amount of data actually written
     */
    int write(byte data[], int off, int n) {
        if(n <= 0) return 0;
```

```
        int remain = n;
        // @todo check if off is valid: 0= <= off < data.length;
throw exception if not


        int i = Math.min(remain, (end < start ? start :
buffer.length) - end);
        if(i > 0) {
             System.arraycopy(data, off, buffer, end, i);
             off    += i;
             remain -= i;
             end    += i;
        }


        i = Math.min(remain, end >= start ? start : 0);
        if(i > 0 ) {
             System.arraycopy(data, off, buffer, 0, i);
             remain -= i;
             end = i;
        }
        return n - remain;
    }




    /**
     * Read as much data as possible from the buffer.
     * @param data   Where to store the data
     * @return       Amount of data read
     */
    int read(byte data[]) {
        return read(data, 0, data.length);
    }


    /**
     * Read as much data as possible from the buffer.
     * @param data   Where to store the read data
     * @param off    Offset of data in array
     * @param n      Amount of data to read
     * @return       Amount of data actually read
     */
    int read(byte data[], int off, int n) {
        if(n <= 0) return 0;
        int remain = n;
        // @todo check if off is valid: 0= <= off < data.length;
throw exception if not


        int i = Math.min(remain, (end < start ? buffer.length : end)
- start);
        if(i > 0) {
             System.arraycopy(buffer, start, data, off, i);
             off    += i;
             remain -= i;
             start  += i;
             if(start >= buffer.length) start = 0;
        }


        i = Math.min(remain, end >= start ? 0 : end);
        if(i > 0 ) {
             System.arraycopy(buffer, 0, data, off, i);
             remain -= i;
```

```
            start = i;
        }
        return n - remain;
    }
}
```

With this ring buffer one can now hand over data from one thread to another in a controlled way. Any other thread-safe, non-blocking mechanism would also do. The key point here is that the write does not block when the buffer is full and also does not block when there is nothing to read.

Using the Buffer together with Serial Events

Usage of OUTPUT_BUFFER_EMPTY Event in Writing

Referring to the skeleton event handler presented in the section Setting up a serial Event Handler[11], one can now use a shared ring buffer from section A simple, thread-safe Ring Buffer Implementation[12] to support the `OUTPUT_BUFFER_EMPTY` event. The event is not supported by all JavaComm implementations, therefore the code might never be called. However, in case the event is available it is one building block for ensuring best data throughput, because the serial interface is not left idle for too long.

The skeleton event listener proposed a method `outputBufferEmpty()`, which could be implemented as it follows.

```
     RingBuffer dataBuffer = ... ;


    /**
     * Handle output buffer empty events.
     * NOTE: The reception is of this event is optional and not
     *       guaranteed by the API specification.
     * @param event The output buffer empty event
     */
    protected void outputBufferEmpty(SerialPortEvent event) {
```

```
    }
```

Reading of Data

The following example assumes that the data's destination is some file. Whenever data becomes available it is fetched from the serial port and written to the file. This is an extremely simplified view, because in reality one would need to check the data for an

11 Chapter 6.2.5 on page 97
12 Chapter 6.2.5 on page 99

end-of-file indication to, for example, return to the modem command mode.

```
import javax.comm.*;


...
InputStream is = port.getInputStream();
BufferedOutputStream out = new BufferedOutputStream(new
FileOutputStream("out.dat"));


/**
 * Listen to port events
 */
class FileListener implements SerialPortEventListener {


    /**
     * Handle serial event.
     */
    void serialEvent(SerialPortEvent e) {
        SerialPort port = (SerialPort) e.getSource();


        //
        // Discriminate handling according to event type
        //
        switch(e.getEventType()) {
        case SerialPortEvent.DATA_AVAILABLE:


            //
            // Move all currently available data to the file
            //
            try {
                 int c;
                 while((c = is.read()) != -1) {
                       out.write(c);
                 }
            } catch(IOException ex) {
                 ...
            }
            break;
        case ...:
            ...
            break;
        ...
        }
       if (is != null) is.close();
       if (port != null) port.close();
    }
```

6.2.6 Handling multiple Ports in one Application

6.2.7 Modem Control

JavaComm is strictly concerned with the handling of a serial interface and the transmission of data over that interface. It does not know, or provide, any support for higher-layer protocols, e.g. for Hayes modem commands typically used to control consumer-grade modems. This is simply not the job of JavaComm, and not a bug.

Like with any other particular serial device, if the control of a modem is desired via JavaComm the necessary code has to be written on top of JavaComm. The page "Hayes-compatible Modems and AT Commands"[13] provides the necessary basic generic information to deal with Hayes modems.

Some operating systems, e.g. Windows or certain Linux distributions provide a more or less standardized way how modem control commands for a particular modem type or brand are configured for the operating system. Windows modem "drivers", for example, are typically just registry entries, describing a particular modem (the actual driver is a generic serial modem driver). JavaComm as such has no provisions to access such operating-system specific data. Therefor, one either has to provide a separate Java-only facility to allow a user to configure an application for the usage of a particular modem, or some platform-specific (native) code needs to be added.

6.3 RxTx

6.3.1 Overview and Versions

Due to the fact that Sun didn't provide a reference implementation of the JavaComm API for Linux, people developed RxTx for Java and Linux `http://rxtx.qbang.org/`. RxTx was then further ported to other platforms. The latest version of RxTx is known to work on 100+ platform, including Linux, Windows, Mac OS, Solaris and other operating systems.

RxTx can be used independent of the JavaComm API, or can be used as a so called provider for the JavaComm API. In order to do the latter, a wrapper called JCL is also needed `http://www.geeksville.com/~kevinh/linuxcomm.html`. JCL and RxTx are usually packaged together with Linux/Java distributions, or JCL is completely integrated into the code. So, before trying to get them separately, it is worth having a look at the Linux distribution CD.

There seems to be a trend to abandon the JavaComm API, and using RxTx directly instead of via the JCL wrapper, due to Sun's limited support and improper documentation for the JavaComm API. However, RxTx's documentation is extremely sparse. Particularly, the RxTX people like to make a mess of their versions and package contents (e.g. with or without integrated JCL). Starting with RxTx version 1.5 RxTx contains replacement classes for the public JavaComm classes. For legal reasons they are not in the `java.comm` package, but in the `gnu.io` package. However, the two currently available RxTx versions are packaged differently:

RxTx 2.0

RxTx version supposed to be used as a JavaComm provider. This one is supposed to have its roots in RxRx 1.4, which is the RxTx version before the `gnu.io` package was added.

RxTx 2.1

RxTx version with a full `gnu.io` package replacement for `java.comm`. This version is supposed to have its roots in RxTx 1.5, where `gnu.io` support started.

13 Chapter 9 on page 121

So, if one wants to program against the original JavaComm API one needs

1. Sun's generic JavaComm version. As of this writing this is in fact the Unix package (which contains support for various Unix versions like Linux or Solaris). Even when used on Windows, the Unix package is needed to provide the generic `java.comm` implementations. Only the part implemented in Java is used, while the Unix native libraries are just ignored.
2. RxTx 2.0 in order to have a different provider below the generic generic JavaComm version than the ones comming with the JavaComm package

However, if one just wants to program against the `gnu.io` replacement package, then

- only RxTx 2.1 is needed.

6.3.2 Converting a JavaComm Application to RxTx

So, if you belong to the large group of people who have been let down by Sun when they dropped Windows support for JavaComm, you are in need to convert a JavaComm application to RxTx. As you can see from the above, there are two ways to do it. Both assume that you manage to install a version of RxTx first. Then the options are either

1. Using RxTx 2.0 as a JavaComm provider
2. Porting the application to RxTx 2.1

The first option has already been explained. The second option is surprisingly simple. All one has to do to port some application from using JavaComm to using RxTx 2.1 is to replace all references to `java.comm` in the application source code with references to `gnu.io`. If the original JavaComm application was properly written there is nothing more to do.

RxTx 2.1 even provides the tool `contrib/ChangePackage.sh` to perform the global replacement on a source tree under Unix. On other platforms such a global replacement is easy to do with IDEs supporting a decent set of refactoring features.

6.4 See also

- Sun Java Communications API[14]
- Java Comm Serial API How-To for Linux[15]
- jSSC - java serial port library. Work under Win32(Win98-Win7), Win64(x86-64), Linux x86, Linux x86-64[16]
- RxTx Home Page[17]
- Unofficial Java Web Start/JNLP FAQ - How can I use Web Start and Comm API together?[18]

14 http://java.sun.com/products/javacomm/
15 http://wass.homelinux.net/howtos/Comm_How-To.shtml
16 http://code.google.com/p/java-simple-serial-connector/
17 http://rxtx.qbang.org/
18 http://lopica.sourceforge.net/faq.html#comm

- SerialIO has a free trial version of their SerialPort package[19]
- Ben Resner has a free download of his SimpleSerial package[20] and a newer version without the C++ code[21]

19 http://serialio.com/products/serialport/serialport.php
20 http://web.media.mit.edu/~benres/simpleserial/
21 http://www.ambientdevices.com/datacasting/index.html

7 Forming Data Packets

Just about every idea for communicating between computers involves "data packets", especially when more than 2 computers are involved.

The idea is very similar to putting a check in an envelope to mail to the electricity company. We take the data (the "check") we want to send to a particular computer, and we place it inside an "envelope" that includes the address of that particular computer.

A packet of data starts with a preamble, some address information, some other transmission-related information, followed by the raw data, and finishes up with a few more bytes of transmission-related error-detection information -- often a Fletcher-32[1] checksum[2]. We will talk more about what we do with this error-detection information in the next chapter, Serial Programming/Error Correction Methods[3].

The accountant at the electricity company throws away the envelope when she gets the check. She already knows the address of her own company. Does this mean the "overhead" of the envelope is useless ? No.

In a similar way, once a computer receives a packet, it immediately throws away the preamble. If the computer sees that the packet is addressed to itself, and has no errors, then it discards the wrapper and keeps the data.

Unfortunately, there are dozens of slightly different, incompatible protocols for data packets, because people pick slightly different ways to represent the address information and the error-detection information.

... gateways between incompatible protocols ...

7.0.1 packet size tradeoffs

Protocol designers pick a maximum and minimum packet size based on many tradeoffs.

- packets should be "small" to prevent one transmitter transmitting a long packet from hogging the network.
- packets should be "small" so that a single error can be corrected by retransmitting one small packet rather than one large packet
- packets should be "large" so more time is spent transmitting good data and less time is spent on overhead (preamble, header, footer, postamble, and between-packet gap).
- the packet header and trailing footer should be short, to reduce overhead

1 http://en.wikipedia.org/wiki/%20Fletcher%27s%20checksum
2 http://en.wikipedia.org/wiki/%20checksum%20
3 Chapter 8 on page 115

- The footer should hold a large error-detection codeword field, because a shorter codeword is more likely to incorrectly accept an error-riddled packet. (We discuss error-detection in more detail in the next chapter, ../Error Correction Methods/[4]).
- making the packet header a little longer, so that meaningful fields fall on byte or word boundaries, rather than highly encoded bit fields, makes it easier for a CPU to interpret them, allowing lower-cost network hardware.
- making the packet header a little longer -- instead of a single error-detection field that covers the whole packet, we have one error-detection field for the header, and another error-detection field for the data -- allows a node to immediately reject a packet with a bit error in the destination address or the length field, avoiding needless processing. The same CRC polynomial is used for both.

- fixed-size packets -- where all packets fall into a few length categories -- do not require a "length" field, and simplify buffer allocation, but waste "internal" data space on padding the last packet when you want to send data that is not an exact multiple of the fixed data size.

7.0.2 start-of-packet and transparency tradeoffs

Unfortunately, it is impossible for any communication protocol to have all these nice-to-have features:

- transparency: data communication is transparent and "8 bit clean" -- (a) any possible data file can be transmitted, (b) byte sequences in the file always handled as data, and never mis-interpreted as something else, and (c) the destination receives the entire data file without error, without any additions or deletions.
- simple copy: forming packets is easiest if we simply blindly copy data from the source to the data field of the packet without change.
- unique start: The start-of-packet symbol is easy to recognize, because it is a known constant byte that never occurs anywhere else in the headers, header CRC, data payload, or data CRC.
- 8-bit: only uses 8-bit bytes

Some communication protocols break transparency, requiring extra complexity elsewhere -- requiring higher network layers to implement work-arounds such as w:binary-to-text encoding[5] or else suffer mysterious errors, as with the w:Time Independent Escape Sequence[6].

Some communication protocols break "8-bit" -- i.e., in addition to the 256 possible bytes, they have "extra symbols". Some communication protocols have just a few extra non-data symbols -- such as the "long pause" used as part of the Hayes escape sequence; the "long break" used as part of the SDI-12[7] protocol; "command characters" or "control symbols"

4 Chapter 8 on page 115
5 http://en.wikipedia.org/wiki/binary-to-text%20encoding
6 http://en.wikipedia.org/wiki/Time%20Independent%20Escape%20Sequence
7 http://en.wikipedia.org/wiki/SDI-12

in 4B5B coding, 8b/10b encoding; etc. Other systems, such as 9-bit protocols,[8][9][10][11][12][13][14] transmit 9 bit symbols. Typically the first 9-bit symbol of a packet has its high bit set to 1, waking up all nodes; then each node checks the destination address of the packet, and all nodes other than the addressed node go back to sleep. The rest of the data in the packet (and the ACK response) is transmitted as 9 bit symbols with the high bit cleared to 0, effectively 8 bit values, which is ignored by the sleeping nodes. (This is similar to the way that all data bytes in a MIDI message are effectively 7 bit values; the high bit is set only on the first byte in a MIDI message). Alas, some UARTs make it awkward,[15][16] difficult, or impossible to send and receive such 9-bit characters.

Some communication protocols break "unique start" -- i.e., they allow the no-longer-unique start-of-packet symbol to occur elsewhere -- most often because we are sending a file that includes that byte, and "simple copy" puts that byte in the data payload. When a receiver is first turned on, or when cables are unplugged and later reconnected, or when noise corrupts what was intended to be the real start-of-packet symbol, the receiver will incorrectly interpret that data as the start-of-packet. Even though the receiver usually recognizes that something is wrong (checksum failure), a single such noise glitch may lead to a cascade of many lost packets, as the receiver goes back and forth between (incorrectly) interpreting that data byte in the payload as a start-of-packet, and then (incorrectly) interpreting a real start-of-packet symbol as payload data.

In order to keep the "unique start" feature, many communication protocols break "simple copy". This requires a little extra software and a little more time per packet than simply copying the data -- which is usually insignificant with modern processors. The awkwardness comes from (a) making sure that the entire process -- the transmitter encoding/escaping a chunk of raw data into a packet payload that must not include the start-of-packet byte, and the receiver decoding/unescaping the packet payload into a chunk of raw data -- is completely transparent to any possible sequence of raw data bytes, even if those bytes include one or more start-of-packet bytes, and (b) since the encoded/escaped payload data inevitably requires more bytes than the raw data, we must make sure we don't overflow any buffers even with the worst possible expansion, and (c) unlike "simple copy" where a constant bitrate of payload data bits results in the same constant goodput of raw data bits, we must make sure that the system is designed to handle the variations in payload data

8 uLan ^{http://ulan.sourceforge.net/} : 9-bit message oriented communication protocol, which is transferred over RS-485 link.

9 Pavel Pisa. "uLan RS-485 Communication Driver" ^{http://cmp.felk.cvut.cz/~pisa/ulan/ul_drv.html} "9-bit message oriented communication protocol, which is transferred over RS-485 link."

10 Peter Gasparik. "9-bit data transfer format" ^{http://www.rtjcom.com/6811/jackpot/rs485-commspec.html#3}

11 Stephen Byron Cooper. "9-Bit Serial Protocol" ^{http://www.ehow.com/facts_7735117_9bit-serial-protocol.html} .

12 "Use The PC's UART With 9-Bit Protocols" ^{http://electronicdesign.com/article/embedded/use-the-pc-s-uart-with-9-bit-protocols6245.aspx} . 1998.

13 Wikipedia: multidrop bus ^{http://en.wikipedia.org/wiki/%20multidrop%20bus} (MDB) is a 9-bit protocol used in many vending machines.

14 ParitySwitch_9BitProtocols ^{http://www.docklight.de/examples_en.htm} : manipulate parity to emulate a 9 bit protocol

15 "Use The PC's UART With 9-Bit Protocols" ^{http://electronicdesign.com/article/embedded/use-the-pc-s-uart-with-9-bit-protocols6245.aspx} . Electronic Design. 1998-December.

16 Thomas Lochmatter. "Linux and MARK/SPACE Parity" ^{http://www.lothosoft.ch/thomas/libmip/markspaceparity.php} . 2010.

bitrate or raw data bit goodput or both. Some of this awkwardness can be reduced by using consistent-overhead byte stuffing.[17] rather than variable-overhead byte stuffing techniques such as the one used by SLIP[18].

Calculate the CRC and append it to the packet *before* encoding both the raw data and the CRC with COBS.[19]

7.1 For further reading

- Optical and radio receivers usually require a preamble of some minimum length in order to synchronize bit clocks. For detailed information on calculating exactly how long (how many transitions) the preamble needs to be, see Clock and Data Recovery/Design values used in practice/Burst transmission mode/Step response of a phase aligner[20].
- `http://intcomm.wiki.taoriver.net/moin.cgi/ProtocolMadness`
- UDP
- Internet Technologies/Protocols[21] including TCP/IP and HTTP
- ATM
- VSCP - Very Simple Control Protocol `http://www.vscp.org/` "The protocol is free"
- "Protocol Design Folklore" by Radia Perlman. Jan 15, 2001. `http://www.awprofessional.com/articles/article.asp?p=20482`
- "Devices that play together, work together: UPnP defines common protocols and procedures to guarantee interoperability among network-enabled PCs, appliances, and wireless devices." article by Edward F Steinfeld, EDN, 9/13/2001 `http://www.reed-electronics.com/ednmag/index.asp?layout=article&articleid=CA154802&spacedesc=readersChoice&rid=0&rme=0&cfd=1`
- CAN bus `http://computer-solutions.co.uk/` `http://computer-solutions.co.uk/gendev/can-module.htm`

"CMX-MicroNet is the first system that allows TCP/IP

and other protocols to be run natively on small processors

... [including] AVR, PIC 18, M16C."

- "byteflight is a high speed data bus protocol for automotive applications" `http://byteflight.com/`
- Nagle's rule ... The Nagle algorithm. "Nagle's rule is a heuristic to avoid sending particularly small IP packets, also called tinygrams. Tinygrams are usually created by interactive networking tools that transmit single keystrokes, such as telnet or rsh. Tinygrams can become particularly wasteful on low-bandwidth links like SLIP. The Nagle

---

17 "Consistent Overhead Byte Stuffing" ^{`http://www.stuartcheshire.org/papers/COBSforToN.pdf`} by Stuart Cheshire and Mary Baker, 1999.

18 `http://en.wikibooks.org/wiki/Serial_Programming%2FIP_Over_Serial_Connections%23SLIP%20`

19 Jason Sachs. "Help, My Serial Data Has Been Framed: How To Handle Packets When All You Have Are Streams" ^{`http://www.embeddedrelated.com/showarticle/113.php`} . 2011.

20 `http://en.wikibooks.org/wiki/Clock%20and%20Data%20Recovery%2FDesign%20values%20used%20in%20practice%2FBurst%20transmission%20mode%2FStep%20response%20of%20a%20phase%20aligner`

21 `http://en.wikibooks.org/wiki/Internet%20Technologies%2FProtocols`

algorithm attempts to avoid them by holding back transmission of TCP data briefly under some circumstances." -- http://www.tldp.org/LDP/nag/node45.html

- The SLIMP3 Client Protocol[22]
- Beej's Guide to Network Programming Using Internet Sockets[23] by Brian "Beej" Hall 2005-11-05
- "RF Link Using the Z86E08"[24] describes yet another "simple" packet protocol ... also mentions a preamble to train the RF receiver just before the rest of the packet.
- Algorithm Implementation/Checksums[25]
- ... other packet protocols ? ...
- Communication Systems/Packet Data Systems[26]
- Communication Networks[27]

22 http://wiki.slimdevices.com/index.php/SLIMP3ClientProtocol
23 http://beej.us/guide/bgnet/output/htmlsingle/bgnet.html
24 http://www.zilog.com/docs/appnotes/an_rflink.pdf
25 http://en.wikibooks.org/wiki/Algorithm%20Implementation%2FChecksums
26 http://en.wikibooks.org/wiki/Communication%20Systems%2FPacket%20Data%20Systems
27 http://en.wikibooks.org/wiki/Communication%20Networks

8 Error Correction Methods

8.1 Introduction

There are 3 main types of handling errors:

- acknowledge or retry (ACK-NAK).
- "Forward Error Correction" (FEC)
- Pretend It Never Happened

8.2 ACK-NAK

Each packet is checked by the receiver to make sure it is "good".

If it *is* good, the receiver (eventually) tells the sender that it came through OK -- it acknowledges (ACK) the packet.

All versions of ACK-NAK absolutely require Two Way Communication[1] .

How does the *receiver* know it's good ?

The sender calculates a checksum or CRC for the entire packet (except for the footer), then appends it to the end of the packet (in the footer/trailer).

The typical CRC is 32 bits, often a Fletcher-32[2] checksum[3].

Aside: Note that the checksum or CRC are forms of **hashing**, ie, irreversibly shrinking data. Checksums and CRCs are weaker algorithms than "cryptographically strong" message authentication code algorithms such as MD5 or SHA variants. Cryptographically strong algorithms can detect errors better than checksums or CRCs, but they take more time to calculate.

Whenever the receiver receives a packet, the receiver calculates exactly the same checksum or CRC, then compares it to the one in the footer/trailer. If they match, the entire packet is (almost certainly) good, so the receiver sends an ACK.

When there's even the slightest question that the packet has any sort of error (which could be *either* in the actual data *or* in the header *or* in the checksum bits -- there's no way

1 http://en.wikibooks.org/wiki/Serial_Programming%3ABi-directional_Communication
2 http://en.wikipedia.org/wiki/%20Fletcher%27s%20checksum
3 http://en.wikipedia.org/wiki/%20checksum%20

for the receiver to tell), the receiver discards it completely and (in most cases) pretends it never saw it.

If it's not good, the *sender* sends it again.

How does the *sender* know it wasn't good ?

It never got the ACK. (So either the packet was corrupted, *or* the ACK was corrupted -- there's no way for the sender to know).

"Stop-and-wait ARQ"

The simplest version of ACK-NAK is "Stop-and-wait ARQ".

The sender sends a packet, then waits a little for an ACK. As soon as it gets the ACK, it immediately sends the next packet. If the sender doesn't hear the ACK in time, it starts over from the beginning, sending the same packet again, until it does get an ACK.

The receiver waits for a packet. If the packet passes all the error-detection tests perfectly, the receiver transmits an ACK (acknowledgment) to the sender.

Subtleties: If the receiver receives the packet perfectly, but the ACK message is delayed too long, then the transmitter sends another copy of the message (a "communication echo"). Imagine the packet contained the message "deduct $11,000 from Fred's account.". When the receiver gets this second copy of the packet, what should it do? Certainly it should send an ACK (otherwise the transmitter will keep trying to send this packet over and over). Either or both of the following problems could occur:

- The delayed first ACK could hit the transmitter after it transmits the second copy of the message, so it transmits the next packet. Then the second ACK hits the transmitter, tricking the transmitter into thinking that "next packet" has been successfully received, when it hasn't.
- When the receiver gets 2 identical consecutive packets saying "deduct $11,000 from Fred's account", are these 2 legitimate independent transactions, and so it should deduct $22,000 from Fred's account? Or is it really just 1 transaction, with a bit of echo, and so should deduct a total of only $11,000 from Fred's account?

Both of these problems can be solved by adding a "sequence number". The transmitter keeps a count of how many independent packets it has transmitted to that receiver, and puts that sequence number in the header of each packet. But when it re-transmits a packet, it re-transmits that same identical packet with that same identical sequence number. Also, the receiver, rather than sending a generic "ACK" message, specifies which particular packet it is responding to by putting its sequence number in the ACK message. When there is a communication echo, the receiver sees the same sequence number, so ACKs that sequence number (again) but then discards and ignores the extra, redundant copy of a packet it already received. When the transmitter is sending a new packet that merely happens to contain the same data, the receiver sees a different sequence number, so it ACKs that new sequence number, and takes another $11,000 out of Fred's account. Poor Fred.

A 1-bit sequence number (alternating 1 - 0 - 1 - 0 for each new packet, and ACK1 ACK0 ACK1 ACK0 in response) is adequate for a stop-and-wait system. But as we will see, other ARQ protocols require a larger sequence number.

Subtleties: Some early protocols had the receiver send a NAK (negative acknowledgment) to the sender whenever a bad packet was received, and the sender would wait indefinitely until it received *either* an ACK *or* a NAK. This is a bad idea. Imagine what happens when (a) a little bit of noise made a bad packet, so the receiver sends the NAK back to the sender, but then (b) a little bit of noise made that NAK unrecognizable. Alternatively, imagine a shared-medium network with 1 sender and 2 receivers. What happens when a little noise messes up the "destination" field of the packet ?

With "Stop-and-wait ARQ", the sender and the receiver only needs to keep 1 packet in memory at a time.

streaming ARQ

The sender sends a packet, then the next packet, then the next, without waiting.

As it sends each packet, it puts a copy of that packet in a "window".

Each packet is consecutively numbered. (The sequence number must be at least large enough to uniquely identify every packet in the window).

... turn-around time ... bouncing off geostationary satellites ...

The receiver occasionally transmits an acknowledgment ("I got all packets up to 8980", "I got all packets up to 8990").

If the receiver is expecting packet number 9007, but it receives a packet with an *earlier* number (that it had already received successfully), it transmits (or possibly re-transmits) a "I got all packets up to 9006" message.

When the sender receives an acknowledgment of any packet in the "window", it deletes that copy.

When the sender's window gets full, it waits a little, then tries re-sending the packets in the window starting with the oldest.

So when the sender suspects an error in some packet, it resend *all* packets starting with the erroneous packet. This guarantees that the receiver will (eventually) receive all packets in order.

Optionally, If the receiver is expecting packet number 9007, but it receives packet number 9008, it may transmit a negative acknowledge (NAK) for 9007, and ignores any higher packet numbers until it gets packet 9007.

When the sender receives a NAK for any packet in the window, it re-starts transmission with that packet (and keeps it in the window).

With "streaming ARQ", the sender needs to keep the entire window of packets in memory at a time. But the receiver still only needs to handle 1 packet at a time, and handles them in consecutive order.

(Some people think of "streaming" as one big packet the size of the window using "stop-and-wait" protocol, divided into smaller "sub-packets").

8.2.1 Selective Repeat ARQ

w:Selective Repeat ARQ[4]

A selective repeat ARQ system is a kind of streaming ARQ.

But instead of the receiver only handling 1 packet at a time, and discarding all packets higher or lower than the one it is looking for, the receiver tries to keep a copy of all packets it receives in a window of its own, and negotiates with the sender to try to resend *only* the erroneous packets.

8.3 FEC

If you have only one-way communication, you are forced to use Forward Error Correction, sometimes called EDAC (Error Detection And Correction).

You transmit the data, then (instead of a CRC) you transmit "check bits" that are calculated from the data.

... NASA space probes ... compact disks ...

The simplest kind is "repeat the message".

If I send the same packet twice, and noise only corrupts one of them, *and* the receiver can tell which one was corrupted, then no data was lost. If I send the same packet 3 times, and noise corrupts any one of them, then the receiver can do "best 2 out of 3". The "check bits" are 2 copies of the data bits. In fact, noise could corrupt a little bit of *all three* of them, and you could still extract all the data -- align the 3 packets next to each other, and do "best 2 out of 3" for every bit. As long as there were only a few bits of noise in each packet, and the noise was in a different place in each packet, all the data can be recovered.

... (put picture here) ...

There are some very clever kinds of FEC (Hamming codes, Reed-Solomon codes) that can correct all kinds of common errors better than "best 2 out of 3", and only require the same number of "check bits" as there are data bits.

8.4 Pretend It Never Happened

A sender often streams audio and video live, in real-time.

What should a receiver do when a packet gets mangled ?

4 `http://en.wikipedia.org/wiki/Selective%20Repeat%20ARQ`

If it sends a message back to the sender, asking it to resend that packet, by the time the reply gets back, it's probably several video frames later. It's too late to use that information.

Rather than pausing the entire movie until the request makes a round-trip, it's far less jarring to the audience if the receiver silently discards the mangled packet, fills in as best it can (for example, with nearby pixels' colors), try not to draw attention to the error, and continue on as if nothing had happened.

Note:
Signal degradation should be documented and easily findable as to let users know that there is no guarantee of exact reproduction.

8.5 combination

Even when they have 2-way communication, sometimes people use FEC anyway. That way small amounts of noise can be corrected at the receiver. If a packet is corrupted so badly that FEC cannot fix it, the protocol falls back on ACK-NAK retransmission (or on Pretend It Never Happened).

8.6 further reading

w:error detection and correction[5]

a detailed description of one ACK-NAK protocol: "XModem / YModem Protocol Reference" by Chuck Forsberg 1988-10-14 `http://www.commonsoftinc.com/Babylon_Cpp/Documentation/Res/yModem.htm`

a detailed description of one streaming protocol: "The ZMODEM Inter Application File Transfer Protocol" by Chuck Forsberg 1988-10-14 `http://www.commonsoftinc.com/Babylon_Cpp/Documentation/Res/zModem.htm`

"Data Link Error Detection / Correction Methods" `http://techref.massmind.org/techref/method/errors.htm` brief descriptions of several error correction methods: Hamming codes, Fire codes, Reed-Solomon codes, Viterbi decoding, etc.

8.7 further reading

- Computer Networks/Error Control, Flow Control, MAC[6]
- Data Coding Theory/Transmission Codes[7]
- Wikipedia:Automatic repeat-request[8] (ARQ)

5 `http://en.wikipedia.org/wiki/error%20detection%20and%20correction`
6 `http://en.wikibooks.org/wiki/Computer%20Networks%2FError%20Control%2C%20Flow%20Control%2C%20MAC`
7 `http://en.wikibooks.org/wiki/Data%20Coding%20Theory%2FTransmission%20Codes`
8 `http://en.wikipedia.org/wiki/Automatic%20repeat-request`

- Wikipedia:forward error correction[9] (FEC)
- Wikipedia:Radio Link Protocol[10]
- On-line CRC calculation and free CRC library[11]
- Algorithm Implementation/Checksums[12]

Category:Serial Programming[13]

9 http://en.wikipedia.org/wiki/forward%20error%20correction
10 http://en.wikipedia.org/wiki/Radio%20Link%20Protocol
11 http://www.lammertbies.nl/comm/info/crc-calculation.html
12 http://en.wikibooks.org/wiki/Algorithm%20Implementation%2FChecksums
13 http://en.wikibooks.org/wiki/Category%3ASerial%20Programming

9 Appendex A:Modems and AT Commands

9.1 Introduction

9.1.1 General

This content is part of the Serial Programming[1] book. It covers the programming of Hayes and Hayes-compatible telephone modems. Such types of modems are the norm in consumer applications, as well as many professionals applications - wherever modems are still used.

Modem programming is slowly becoming a lost art, particular with the wide-spread movement of users from modem dial-up lines to DSL for very obvious performance reasons. Still modems are used for many applications, at home, or in a professional environment. In recent times, modems can be found in new areas where they were previously not seen. E.g. embedded modems in machines are used to automatically "call home" to the manufacturer in case the machine is in need of some service. Often this is done via a wireless phone system, where the wireless module still provides a Hayes-compatible interface for dialing and data transmission.

The original Hayes modem command set is exclusively used as a reference in this module. **Vendor specific extensions are not covered, and do not belong into this module**. The module explains the origin of the term *Hayes*, and the related *AT commands*. Also some principal information about what a *modem* is, and how the signaling with a modem happens are provided for completeness. The module then continues with a description of the basics of modem programming, including the set-up of a development environment.

Further, the content provides detailed programming information (*incomplete*), and a reference of the original Hayes command set and registers (*incomplete*).

9.1.2 Administrative Information

This section particularly addresses potential authors. Please note:

- This module **is not** a dumping ground for random modem programming information and folklore.
- This module is **operating system agnostic**. The Programming Serial Data Communications[2] book provides other modules for such information.

1 http://en.wikibooks.org/wiki/Serial%20Programming
2 http://en.wikibooks.org/wiki/Programming%3ASerial%20Data%20Communications

- This module deals with **generic Hayes modems**, not with any vendor specific extensions. If you really want to see your particular love-child covered, provide an Appendix with that vendor/brand specific information.
- Do not assume that just because something works on you particular modem it is the standard and other modems do it the same way. If you have no first hand experience that something is done the same way on "almost" all Hayes-compatible modems, then leave it out, or mark it at least as doubtful.

The reason why this module sticks with the original Hayes command set is to have a defined boundary. This module is not intended as a reference manual. Once someone has mastered the basic set, and implemented the code, it is rather straight forward to deal with vendor-specific extensions. Other extensions, e.g. the very rough and basic FAX extensions require some deep insight into the involved protocols (e.g. in the case of FAX the detailed encoding, compression and timing of fax data on the phone line). This is out of the scope of this book. If you know how to handle the FAX extensions, write your own book.

9.1.3 What is Hayes?

Hayes Microcomputer Products, Inc. was a modem manufacturer from the beginning of the 1980s until the end of the 1990s, with its heyday in the early '90s. The name *Hayes* still exists as a brand name, owned by *Zoom Telephonics, Inc.* (as of Fall 2004).

In 1981, Hayes developed the **Hayes Smartmodem**. This was a unique product at the time, because this modem was no longer simply a "dumb" device blindly converting serial data to and from audio tones, but contained some "intelligence". It was possible to send commands to the modem to configure it, to execute certain operations (such as dialling a number, quieting the speaker, hanging up, etc.), and to read the current status of the connection. Hayes developed and published a command set to control the modem over a serial line. This command set became popular among consumer modem manufacturers, and was cloned a thousand times. Known as both the "Hayes command set" and the "AT command set", it has long been the de-facto standard for controlling consumer modems and also many professional modems. Modems which support this command set are called *Hayes-compatible*.

The commands were standardised at some point in time, however, as it is typical with standards, there are several standards. Plus, of course, there are still vendor-specific extensions and implementations in different modems vary slightly. Some of these enhancements were required to support at that time emerging features, such as data compression and FAX support. As a result, the command sets of modern modems are not fully compatible with each other. The original Hayes commands, however, should still work, and still form the core of almost all consumer modem command sets.

The basic set of commands was at some point in time standardised as TIA/EIA-602[3] and the syntax as EIA/TIA-615. But as already mentioned, modem manufacturers added their extensions. A larger extended set, particular under the pressure from cell phone manufacturers, was standardised as ITU V.250[4] (old name V.25ter). That one usually

3 http://www.tiaonline.org/standards/search_results2.cfm?document_no=TIA/EIA%2D602
4 http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-V.250

forms the base for professional Hayes-compatible modems, and cell phones with build in data modems. ITU V.250 further referes to a bunch of other standards (e.g. V.251, V.252, V.253) for particular applications and extensions, and also has some supplements. Plus, of course there are the many standards defining other aspects of a modem, like compression and transmission.

See Also:

- Wikipedia:Hayes Communications[5]
- Transferring Data between Standard Dial-Up Modems[6]

9.1.4 What are AT Commands?

Almost all of the Hayes modem commands start with the two letter sequence `AT` - for getting the modem's *attention*. Because of this, modem commands are often called *AT Commands*. This still holds for many of the manufacturer specific command set extensions. Most of them also start with `AT`, and are called *AT Commands*, too. Please note, that just because an AT command contains a *&* does not make it an extensions. *&* commands were already part of the original Hayes command set.

The exact usage of the term *AT command set* slightly varies from manufacturer to manufacturer, often subject to marketing blurbs. In general, it can be assumed that a modem with an *AT command set*

- uses commands mostly starting with `AT`,
- uses the original Hayes way of separating data and commands, and
- supports the original Hayes commands and register settings as a subset.

9.1.5 What is a Modem?

w:Modem[7] A modem in the classic sense is a **mo**dulator/**dem**odulator for transmitting digital information over analog wires, such as the analog telephone system's two-wire or four-wire lines. The term has come to be used as acceptable slang for many communication devices used to link a computer to either another computer, or a wide-area network (Wikipedia:WAN[8]). For example, the Ricochet radio data transceivers were commonly known as "Ricochet modems".

This module deals with the classic type of *smart* modems, designed to convert data from/to a serial interface to/from an analog line. The module also applies to modems which provide the classic serial interface but connect over a different physical layer, such as a digital line, as well as devices providing a serial modem-like interface for other purposes. For our purpose, the modem is a classic DCE (data communications equipment) device, controlled via serial line by a classic DTE (data terminal equipment) device (such as a computer).

5 http://en.wikipedia.org/wiki/Hayes%20Communications
6 http://en.wikibooks.org/wiki/Transferring%20Data%20between%20Standard%20Dial-Up%20Modems
7 http://en.wikipedia.org/wiki/Modem
8 http://en.wikipedia.org/wiki/WAN

Depending on the type of modem, the modem can use a number of different technologies and speeds to transmit the data over the analog line. The details of these technologies are of no particular interest here, other than to note that it is possible with most modems to specify these communication parameters (for example, to disable compression, or to change modulation techniques). The data this module deals with is not the data on the analog line, but the data as it appears on the serial interface between the DTE and DCE. I.e. the data as read and written by a device like a computer.

(*Smart*) Modems also provide auxiliary services, such as dialling a particular number to set up a connection. As a consequence, a modem can be in a number of different states and modes, which are not always orthogonal. It is possible, for example, for a modem to be in the command mode while still keeping a connection (see the +++ sequence for details).

Non-smart modems had to rely on other equipment like an ACU (automatic call unit) to provide these auxiliary services, but they are practically extinct today.

9.1.6 Inband Signalling

The original RS232C/V.24 specification contained a TX wire for transmitting data and a RX wire for receiving data, and other completely separate wires for transmitting control information between the DTE and DCE, the idea being to separate data and control information. In telecommunication jargon this is called **outband signalling**.

Hayes-compatible modems use almost none of these RS232C/V.24 features. Instead, communication with the modem is done almost exclusively via the same RX/TX lines which are used for transferring the data. This mechanism is called **inband signalling**.

Inband signalling has significant disadvantages. At any point in time, both the DTE and DCE must know if information sent or received via the TX and RX lines is for signalling purposes, or if it is data, which should be handled transparently. Therefore, the DTE and DCE must operate in sync. If they get out of sync, either data will be lost, data will be incorrectly interpreted as commands, or signalling information will be interpreted as data, effectively destroying the original data.

Inband signalling has the advantage that the wiring between the DTE and DCE is simpler, and also that, at least at first glance, the communication software in the DTE is simpler.

As it has been said, Hayes-compatible modems use almost none of the RS232 control lines. But only almost. For example, they often drive DCD (data carrier detect). This, however creates the situation that modem-driving software now has to take care not only of the inband, but also the outband signalling with a modem. This slightly complicates the communication software's state machine[9].

Further, especially with the rise of cell phone modems, manufacturers have again started to introduce more outband signaling. Such modems provide multiple virtual serial interfaces. Some of these interfaces are exclusively dedicated to data transport, controlled by another serial interface which is either used exclusively for signalling (i.e. outband signalling) or

9 Chapter 9.2.6 on page 131

can still also be used in the more conventional inband signalling scenario. In such cases the communication software needs to manage even more complex states.

9.1.7 Command State / On-line State

With respect of controlling the modem a Hayes-compatible modem is one of two main states:

Command State

The modem interprets data from the DTE as modem commands. The modem can be in command state while still keeping a connection with a remote party.

On-line State

The modem interprets data from the DTE as payload and transmits it to the other party. This state requires that a connection to the remote site has been established.

Inside these main states are a number of sub states. Also, with respect to other issues a modem has a number of communication states, e.g. if a remote carrier has been detected or not.

9.1.8 Originating Mode / Answer Mode

Originating mode

A modem in originating mode is a modem which is setting up a connection, e.g., by dialing the number of a remote station and initiating the negotiation of protocols.

Answer Mode

A modem in answer mode is a modem waiting to be contacted and ready to "answer the phone".

9.1.9 Command Responses

A modem is supposed to send a response for almost all commands it receives. These responses can either be in the form of ASCII strings, or numeric values. The response type can be switched with a command, but it is typical to use the ASCII responses.

Responses need to be tracked by the DTE with great care. Among other things they inform the DTE if the dialling of the remote site was successful or not, and if the modem switches from command state to on-line state or not.

Unfortunately, the set of response messages has been greatly enhanced since the original Hayes modems and are often configurable via additional `AT` commands. It is suggested to not strictly parse response messages but to forgivingly check if they contain interesting keywords, like `CONNECT`. It is also suggested to study the manual of a particular modem very carefully.

9.1.10 S-Registers

The so called S-registers are also a Hayes heritage which all Hayes-compatible modems support. They are registers in the modem which contain various settings. And like the AT commands, they have been extensively enhanced by different modem manufacturers.

The reason why they are called **S**-Registers is a little bit unclear. Some say the **S** stands for modem *settings*. Some say they are just called like this, because they are set and read with `ATS`... commands. In the common vernacular they were usually termed *storage* registers because they permanently stored the values even through power-off.

Several of the other `AT` commands also change values of particular S-Registers. There is usually no difference in setting a value directly via an S-Register or via another `AT` command. It depends on the particular situation which way of setting a register is better.

9.2 Modem Programming Basics

9.2.1 Command Reference

In order to program for an actual modem it is a rather good idea to obtain the command reference for that particular modem. Unfortunately, it has become quite common for no-name modems to ship without any kind of usable command reference. Thanks to Windows' Plug & Play feature it is no longer necessary on Windows to know the individual commands. Instead, all that is needed for a modem to run on Windows is to be shipped with the necessary `.inf` files (often hidden inside some "installer" software, and called a "driver" which is technically not the case, Windows already contains the necessary drivers).

If the modem doesn't come with a command reference the next logical step is to search the web. However, unfortunately, a lot of modem information has vanished from the surface of the earth and the web in recent years. With the rise of broadband Internet connections, modems have become old fashioned devices and many sources are no longer available. It has become more and more difficult to find basic information about particular modem types. Even for modern modems like cell phone modems it can be difficult to find the necessary information.

There are a number of alternatives to obtain a command reference if one doesn't come with the modem:

- Maybe the distributor provides one on its website
- Maybe the OEM manufacturer provides one.
  This requires to identify the OEM manufacturer. A possible way is to use the FCC number of the device, and then looking the original manufacturer up on the FCC web site.
- Maybe the chipset manufacturer provides one.
  Consumer modems are often just build around "off-the-shelf" modem chipsets from larger hardware manufacturers. The cheaper the modem, the more likely it is that the modem manufacturer didn't change anything in the firmware and is using the original example software from the chipset manufacturer. Some chipset vendors provide command references for their modems.

- By looking into the corresponding Windows `.inf` files it is possible to at least obtain the basic commands
- By using the generic Hayes command reference in this Wikibook module.
- Obtaining the previously mentioned standard documents if there is an indication a particular modem complies to such a command standard.
- Using some kind of *sniffer* program to monitor the communication between the modem and the DTE and reverse engineering the commands using the obtained information. This requires that (a) reverse engineering is legal in your justification and (b) that there is some DTE communication software available that handles the particular modem so there is some valid communication to sniff.

9.2.2 Setting up a Development Environment

It is highly recommended to spend some preparation time setting up a suitable development environment before starting to write drivers or software for a modem. Most of this consists of hardware set-up.

It is suggested to set up a small network with a "remote" computer and a second modem in answer mode. "Remote" computer in this case means a computer sitting right next to the development machine, but connected via the modems. If a **terminal program** is being developed, the "remote" computer should run some small BBS software (for example), so there is always someone ready to answer, and/or protocol analysis/data dump software. Developing modem software without such a setup can be extremely frustrating. Such a set-up pays off a hundred times in reduced development time and lower stress. Likewise, the modems used should have real speakers, and support `ATM`*n* commands well enough that you can leave the speaker on for the entire connection process (and ideally have the option to leave it on, period). "Debugging by ear" can be a reality with modems, particularly during compatibility testing.

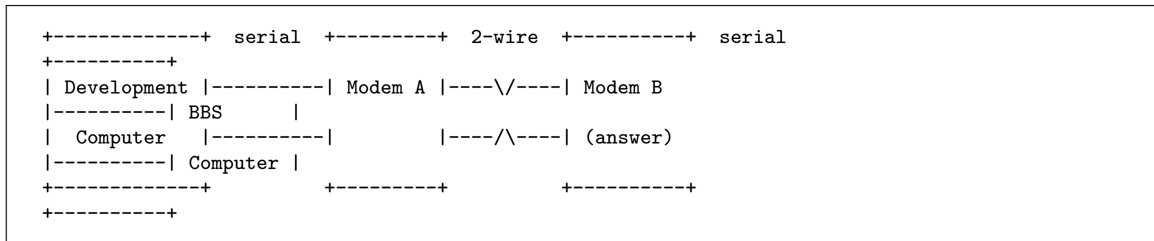
If possible, a hardware protocol analyser, or at least an RS-232 breakout box[10], should be obtained. These can be placed between the computers and modems, if needed, to troubleshoot the serial link and ensure that data is, in fact, being transferred between the modem and the computer -- a sanity check which comes in handy far more often than you might expect. Actual hardware protocol analysers are surprisingly expensive, however; old Wyse terminals are not, and are almost as useful for this purpose. If you find one, pick it up. Terminals that support automatic baud-rate detection are particularly useful.

If dialing with the modem also needs to be tested, a small analog PABX for home usage is needed. These PABX units are dirt cheap; an analog PABX for four internal lines and one external line should cost no more than US$50. If dialing is not needed, then the modems should be capable of directly driving a two-wire or four-wire line in **leased-line** mode; otherwise, the PABX is still needed.

Possible setups are for example:

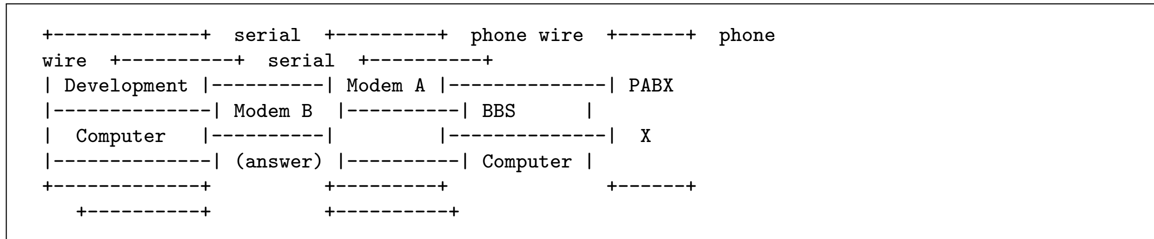
a) Leased-Line Mode

10 Chapter 2.2.5 on page 10



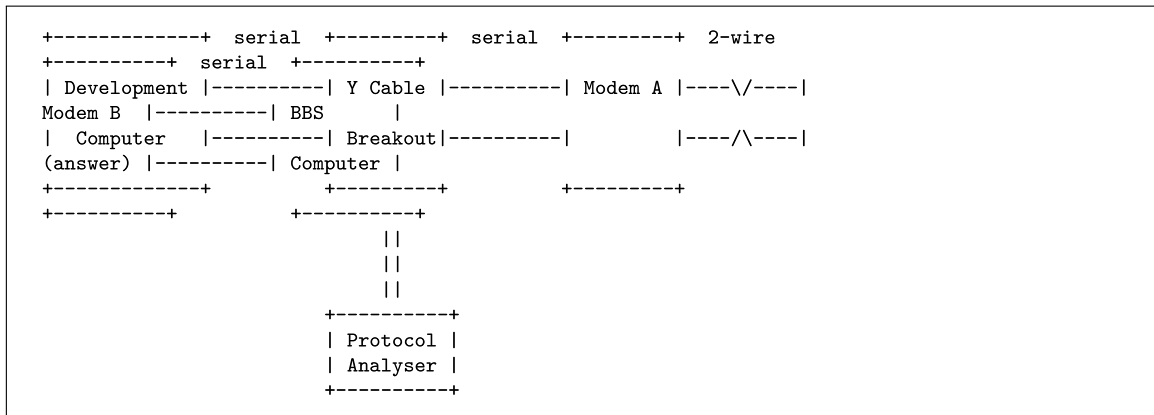
or

b) With PABX



or

c) Leased-Line Mode with Protocol Analyser



Other combinations are of course also useful. And being able to easily reconnect the protocol analyser, e.g. between Modem B and the BBS Computer is helpful, too.

9.2.3 Operating System, Programming Language & Communication Basics

Before dealing with the details of handling a modem, a few basics should be in place. First of all, the communication with the serial interface should be in place. This includes that the APIs as provided by the particular operating system for serial communication - if any - should be understood. If the operating system doesn't provide such APIs, then it is recommended to first implement the UART access and wrap it into a library, if the serial UART in some hardware is supposed to be programmed directly. Alternatively, a programming language which provides convenient access to a serial interface can be used.

Whatever is used, it should be tested before starting to program for the modem. There is nothing more annoying than not knowing if a particular misbehaviour is caused by a failure in the serial communication with the modem, or is a problem with the modem (usually with the commands sent to it).

Unless in the most simple case, it is suggested to use hardware handshaking with the modem - particularly for speeds greater then 2400 bps or 9600 bps. Therefore, the used low-level serial communication software and hardware should support hardware handshake. If the UART supports some FIFO, like the 16550 UART, the FIFO should be enabled (both for sending and receiving data).

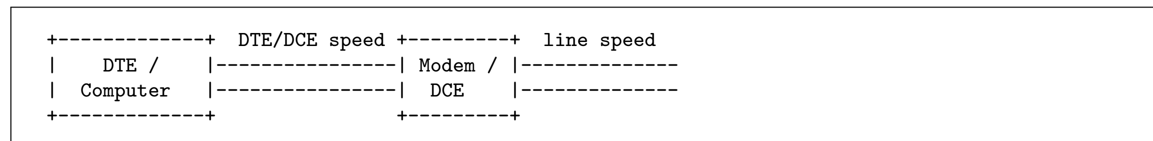
It is undecided if data reception via polling or via interrupts is better. If every incoming byte raises an interrupt there are many interrupts at high communication speeds, and, as surprising as it might sound, polling the UART might be more efficient in such cases.

Communication as supported by a modem is usually half-duplex. Either the DTE or the DCE talks, the other side is supposed to listen. The communication with the modem should best be done with

- 8 Bit
- No parity
- 1 Stop bit

See the next section for speed information.

9.2.4 Line Speed is not DTE/DCE Speed



Helpful Hint:
Some modem manufacturers call the DTE/DCE speed *DTE speed*, and the line speed *DCE speed*. Others distinguish between *DTE speed* (DTE/DCE speed on the serial interface), *DCE speed* (bps between the modems), and *line speed* (Baud rate between the modems). Carefully observing the terminology can help to correctly interpret a manufacturer's documentation.

An issue which can be very confusing is the difference between the line speed (the data transfer speed on the telephone line) and the speed on the serial line between the DTE (computer) and the DCE (modem).

First, there is always some general confusion about the line speed, because some line speed is given with taking compression into account, while other data is given without taking compression into account. Also, there is a difference between *bps* and *Baud* due to the modulation schema used on the line. In addition, marketing blurbs obscure the picture. We will not make any attempt to clean up the long-standing Baud vs. bps confusion here (it is hopeless :-)). It is just recommended that whenever the modem returns information about line speed the above mentioned differences are taken into account to avoid any misinterpretation.

Second, the speed on the telephone line does not necessarily have to be the same as the speed on the serial line. In fact, it usually isn't on modern modems. It is recommended to set the DTE/DCE speed to a fixed speed instead of following the line speed. Logically, the fixed DTE/DCE speed should be large enough to cope with the highest expected line speed. V.90 modems should e.g. be accessed via 115200 bps or higher on the serial interface.

Setting the DTE/DCE speed on modern modems is quite simple. They all use autosensing on the serial interface. That is, they themselves detect the speed of data as received from the DTE and use the same speed to return data to the computer. They usually also autosense the parity, and 7 bit / 8 bit data length. Usually modems assume one stop bit when autosensing the serial interface. Therefore it is enought to just configure the serial interface on the DTE to the desired DTE/DCE communication parameters and let the modem figure it out on its own.

Autosensing can fail in rare cases and some modems might have broken autosensing. If a modem tends to fail autosensing it can help to start the initial communication after the DTE is configured with one or more *nop* AT commands

```
AT<CR>
```

repeated a limited number of times until the modem starts to return

```
OK
```

for the *nop* commands.

When a modem sets up a connection with a remote party it can report the used speed. In fact, it can report the line speed or just the DTE speed (some modems can report both). The end user is most probably interested in the line speed, and not the DTE/DCE speed. So from this point of view, it is best to set the modem to report the line speed, and e.g. write the received information to a log file. However, some old communication software or modem drivers interpret the response from the modem as a request to change the DTE/DCE speed. In such cases the modem must be set to always return the DTE/DCE speed. Since this DTE/DCE speed will be the same as detected via autosensing there will be no speed change.

In the rare case that the DTE/DCE speed should indeed follow the line speed, the responses from the modem should of course be set to return the line speed. Then the DTE software has to evaluate the response, and change the DTE/DCE speed accordingly. This is really not recommended these days.

See the #W: Negotiation Progress Message Selection[11] command for details on how to set which response to get.

11 Chapter 9.6 on page 134

9.2.5 Character Set and Character Case

Commands sent to the modem, and textual responses are supposed to be in the ISO 646 character set. ISO 646 is just another name for the familiar 7-bit ASCII[12] character set. Typically, modems chop off any 8th bit in commands they receive anyhow. They interpret the result as if the command has been sent using only 7-bit characters. However, it is not recommended to rely on this, but instead ensure that commands are only sent using 7-bit characters.

Commands are not case sensitive, assuming a modern modem. Some early modems insisted on uppercase-only commands. Still, a generic driver could do worse than ensuring that all commands are sent in uppercase, and all responses are interpreted case-independent. Typically, both letters of the `AT` command prefix must be of the same case. So `AT` and `at` are acceptable, while `At` and `aT` are not.

9.2.6 Welcome to the World of State-Machines

Modem programming means to tap into the world of telecommunications. This is an unknown field for most amateur, as well as professional programmers. Telecommunication is heavily centered around state-machines. And in fact, it is rather difficult or impossible to program a modem without using a state-machine. The modem is at any time in a particular state, and any DTE software which tries to control and use the modem needs to track the state of the modem - in an own state machine. This is necessary, because a Hayes-compatible modem can only do certain things when it is in a certain state. E.g. it can only dial out if it is not already connected to some remote site.

Part of a modem's state can be tracked via particular RS-232 lines. E.g. DCD (data carrier detect) can be used to figure out if the modem has detected a remote modem's carrier signal. Other information is provided by the flow-control lines. However, some states, and associated data need to be tracked via interpreting the modem's result codes[13].

People unfamiliar with the theory and practice of state machines often try to circumvent the issue by "tough coding". Which means, they throw more and more code onto the problem (wrapped in a heap of if/the/else/otherwise/maybe/... statements), until things seem to work - sort of. If they are lucky they have implicitly managed to create a state machine which works. If they are unlucky, they end up with a partial state machine, which breaks down should something unusual happen in the communication. This usually comes with the problem that the software was not designed to recover if things break down. So such software tends to hang or crash.

It is much more efficient to first spend a few hours to to learn the basics of simple state machines, and then spending a few more hours to describe the communication with the modem as a state machine. The result of this planning serves as a nice template for implementing the DTE software.

12 `http://en.wikipedia.org/wiki/ASCII`
13 Chapter 9.8 on page 137

9.3 Flow Control

A slow device needs a way to tell its peer that currently, it is busy, so further incoming data must be stopped until this slow device tells otherwise. This mechanism is provided by flow control. There are two ways of doing flow control: by hardware or software.

9.3.1 Hardware Flow Control

Hardware flow control is usually implemented using the CTS (Clear To Send[14]) and RTS (Request To Send[15]) lines, which needs separate hardware data lines between devices. This is allocated in the RS-232 cable specification.

Hardware flow control based on DSR (Data Set Ready[16]) and DTR (Data Terminal Ready[17]) is uncommon, particular for modems. It can usually be found at serial printers. Again, DSR/DTR hardware flow control requires additional hardware data lines between devices.

From a programming point of view there is usually not much difference in programming CTS/RTS or DSR/DTR hardware flow control. The hardware has to provide means to drive/read the corresponding signals in the serial interface. If the hardware supports both, CTS/RTS and DSR/DTR flow control, then it is recommended to support both and provide the user with a configuration option.

It should be noted that some hardware or operating system drivers do not provide means to drive/read the less common DSR/DTR combination. If the remote device insists on DTR/DSR flow control a common workaround is to use CTS/RTS in the software, but rewire the cabling so the CTS/RTS wires are in fact connected to DSR/CTS.

9.3.2 Software Flow Control

This kind of flow control doesn't need extra signal line(s) like hardware flow control, but instead uses special control characters within the data content. To stop further incoming data, the receiving device sends the XOFF character. To enable more data, an XON character will be sent.

However, since the data being sent cannot contain these characters (unless you know that the receiving device ignores such information), binary (non-ASCII) data cannot be transmitted this way. Software flow control is typically used for communications to terminals and other character-based devices. Binary data should not be sent this way as it could, randomly, contain these characters. Hardware flow control using RTS/CTS is usually used.

Helpful Hint: Realizing that the Control Key is a special "shift" key that chops off the 100 bit (octal), it is easy to remember that the ASCII character used for sending XOFF is a Control-S (23 Octal) while the character for XON is a Control-Q (21 Octal). [Think of "S" for Stop and "Q" for Qontinue... don't you spell it that way?]

14 Chapter 2.3.3 on page 14
15 Chapter 2.3.3 on page 14
16 Chapter 2.3.3 on page 14
17 Chapter 2.3.3 on page 14

9.4 Changing State

9.4.1 General

Changing the state from command state to on-line state or vice versa is either straightforward or a great mystery. This module covers the more obscure ways.

9.4.2 On-line State to Command State

It is of course possible to switch from on-line state to command state by dropping the connection (going on-hook in modem terminology). It is also possible to temporarily switch into command state while keeping the connection.

Going on-hook programmatically (and not via dropping a modem control line) requires to first switch into command state while keeping the connection, too.

Switching into command state, while in fact in the middle of transferring data (nothing else is meant with on-line state) requires to send a certain escape sequence as part of the data. This escape sequence is detected by the modem and the modem changes state. Since this character sequence might also be part of the normal data, an additional mechanism is needed to separate the escape sequence from normal data. This is the curse of inband signalling.

The separation of the escape sequence is done by using a so called guard time, which was once patented by Hayes. As a result, some modem manufacturers eliminated the guard time using an alternate escape sequence called the Time Independent Escape Sequence. Anyway, the escape sequence is only recognized by the modem when there was no other data from the DTE (terminal) for at least the duration of the guard time, and when there was no other data from the terminal after the escape sequence for at least the duration of the guard time, too.

An escape sequence consists of three times the same particular character. The character, as well as the guard time is configurable. By default, the character is `+`, and the guard time is one second. So, with the default configuration, a change to command state requires

```
<1 sec. nothing>+++<1 sec. nothing>
```

If the connection should be dropped, this escape sequence should be followed by the `AT` command to go on-hook, which is `ATH0`:

```
<1 sec. nothing>+++<1 sec. nothing>ATH0<CR>
```

9.4.3 Command State to On-line State

The usual way to go from command state to on-line state is via dialing the remote site (see `D` command). But if the connection already exists, and the modem has been switched to command mode via the escape sequence, the way is different.

If the connection should not be dropped, but instead data transmission should be continued, the `ATO0` (letter o, digit zero) command is needed:

```
<1 sec.  nothing>+++<1 sec.  nothing>
send a few more modem commands, then go back on-line
ATO0<CR>
```

9.5 Sync. vs. Async. Interface

9.6 X.25 Interface

9.7 AT Commands

The following list is the list of the original Hayes commands. Different modems use slightly different commands. However, this list is supposed to be as "generic" as possible, and should not be extended with modem specific commands. Instead it is recommended to provide such command lists in an Appendix.

9.7.1 AT Command Format

Here is a summary of the format and syntax of AT commands. Please note that most of the control characters are configurable, and the summary only uses the default control characters.

- AT commands are accepted by the modem only when in command mode. The modem can be forced into command mode with the #+++: Escape Sequence[18].
- Commands are grouped in command lines.
- Each command line must start with the #AT: Command Prefix[19] and terminated with #<CR>: End-of-line Character[20]. The only exception is the #A/: Repeat Last Command[21] command.
- The body of a command line consists of visible ASCII characters (ASCII code 32 to 126). Space (ASCII code 32) and ASCII control characters (ASCII code 0 to 31) are ignored,

18 Chapter 9.6 on page 134
19 Chapter 9.6 on page 134
20 Chapter 9.6 on page 134
21 Chapter 9.6 on page 134

with the exception of #<BS>: Backspace Character[22], #<CAN>: Cancel Character[23], and #<CR>: End-of-line Character[24].

- All characters preceding the #AT: Command Prefix[25] are ignored.
- Interpretation / execution of the command line starts with the reception of the first (and also command-line terminating) #<CR>: End-of-line Character[26].
- Characters after the initial #AT: Command Prefix[27] and before the #<CR>: End-of-line Character[28] are interpreted as commands. With some exceptions, there can be many commands in one command line.
- Each of the basic commands consists of a single ASCII letter, or a single ASCII letter with a `&`prefix, followed by a numeric value. Missing numeric values are interpreted as `0` (zero).
- The following commands can't be followed by more commands on the command line. They must always be the last commands in a command line. If they are followed by other commands, these other commands are ignored. However, some of these commands take command modifiers and it is possible that a following command is accidentally interpreted as a command modifier. Therefore, care should be taken to not follow these commands with any more commands on the same command line. Instead, they should be placed in an own command line.
  - #A: Answer Command[29]
  - #D: Dial Command[30]
  - #Z: Soft Reset Command[31]
- A command line can be edited if the terminating #<CR>: End-of-line Character[32] has not ben entered, using the #<BS>: Backspace Character[33] to delete one command line character at a time. The initial #AT: Command Prefix[34] can't be edited/deleted (it has already been processed, because upon reception of the #AT: Command Prefix[35] the modem immediately starts command line parsing and editing, but not execution).
- The modem echoes command lines and edits when #E: Command State Character Echo Selection[36] is on (surprise, surprise :-)).

22 Chapter 9.6 on page 134
23 Chapter 9.6 on page 134
24 Chapter 9.6 on page 134
25 Chapter 9.6 on page 134
26 Chapter 9.6 on page 134
27 Chapter 9.6 on page 134
28 Chapter 9.6 on page 134
29 Chapter 9.6 on page 134
30 Chapter 9.6 on page 134
31 Chapter 9.6 on page 134
32 Chapter 9.6 on page 134
33 Chapter 9.6 on page 134
34 Chapter 9.6 on page 134
35 Chapter 9.6 on page 134
36 Chapter 9.6 on page 134

- When echo is on, #<BS>: Backspace Character[37]s are echoed with a sequence of <BS> <BS> (backspace, space, backspace) to erase the last character in e.g. a terminal program on the DTE.
- A command line can be cancelled at any time before the terminating #<CR>: End-of-line Character[38] by sending the #<CAN>: Cancel Character[39]. No command in the command line is executed in this case.
- The #A: Answer Command[40] and #D: Dial Command[41] can also be cancelled as long as the handshake with the remote site has not been completed. Cancellation is done by sending an additional character. In theory, it doesn't matter which character. But care has to be taken that cancellation is not attempted when the handshake has already completed. In this case the modem has switched to on-line state (#Command State to On-line State[42]) and the character will be send to the remote side. A save way to avoid this problem is to always use the #+++: Escape Sequence[43] followed by going on-hock with the #H: Hook Command Options[44]. If the modem is already in the on-line state, this will drop the connection. If the modem is still in the handshake phase the first character of the #+++: Escape Sequence[45] will cancel the command (and the rest will be interpreted as a normal command line, doing no harm).
- Command line execution stops when the first command in the command line fails, or the whole command line has been executed. Every command before the failed command has been executed. Every command after the failed command and the failed command in the command line has not been executed.
- There is no particular indication which command in a command line failed, only that one failed. It is best to repeat the complete command line, or to first reset the modem to a defined state before recovering from a failure.
- A modem only accepts a new command line when the previous command line has been executed (half-duplex communication). Therefore, care should be taken to only send the next command line after the result code from the previous command line has been received.

9.7.2 *Command Description Template*

To be removed when all commands are documented.

Syntax:

37 Chapter 9.6 on page 134
38 Chapter 9.6 on page 134
39 Chapter 9.6 on page 134
40 Chapter 9.6 on page 134
41 Chapter 9.6 on page 134
42 Chapter 9.4.3 on page 134
43 Chapter 9.6 on page 134
44 Chapter 9.6 on page 134
45 Chapter 9.6 on page 134

```
<The syntax of the command, when necessary in EBNF>
```

Description:

<Description of the command, including information about the purpose and effects>

Result Codes:

Result Codes

Code	Description
OK	Parameter was valid *<description of success>*
ERROR	Otherwise *<description of failure>*

Related Commands and Registers:

- *<Link list of related commands and registers>*

9.7.3 Special Commands and Character Sequences

See Special Commands and Character Sequences Reference[46]

9.7.4 AT Commands A - M

See AT Commands A - M[47]

9.7.5 AT Commands N - Z

See AT Commands N - Z[48]

9.7.6 AT& Commands

See AT& Commands[49]

9.8 Result Codes

See Result Codes[50]

46 http://en.wikibooks.org/wiki/Serial%20Programming%2FModems%20and%20AT%20Commands%2FSpecial%20Commands%20and%20Character%20Sequences
47 http://en.wikibooks.org/wiki/Serial%20Programming%2FModems%20and%20AT%20Commands%2FCommands%20A%20-%20M
48 http://en.wikibooks.org/wiki/Serial%20Programming%2FModems%20and%20AT%20Commands%2FCommands%20N%20-%20Z
49 http://en.wikibooks.org/wiki/Serial%20Programming%2FModems%20and%20AT%20Commands%2F%26%20Commands
50 http://en.wikibooks.org/wiki/Serial%20Programming%2FModems%20and%20AT%20Commands%2FResult%20Codes

9.9 S-Registers

See S-Registers[51]

9.10 Advanced Features

9.10.1 Introduction

Modern consumer modems provide a number of additional features which were originally uncommon for a modem, but became standard features over time. This section provides an overview about how to program these features.

9.10.2 Fax Class 1

9.10.3 Fax Class 2

9.10.4 Voice Services

51 http://en.wikibooks.org/wiki/Serial%20Programming%2FModems%20and%20AT%20Commands%2FS-Registers

10 Contributors

Edits	User
10	Adrignola[1]
2	Alsocal[2]
2	Benoswald[3]
1	Boots8181[4]
10	Breakpoint[5]
10	Dallas1278[6]
1	Damian Yerrick[7]
10	Darklama[8]
30	DavidCary[9]
2	DavidL[10]
1	Derbeth[11]
3	Dirk Hünniger[12]
3	EdDavies[13]
1	Fishpi[14]
3	Geocachernemesis[15]
2	Guanabot[16]
1	Gumba gumba[17]
1	HumbertoDiogenes[18]
1	Insaneinside[19]
2	JenVan[20]
10	Jguk[21]

1 http://en.wikibooks.org/w/index.php?title=User:Adrignola
2 http://en.wikibooks.org/w/index.php?title=User:Alsocal
3 http://en.wikibooks.org/w/index.php?title=User:Benoswald
4 http://en.wikibooks.org/w/index.php?title=User:Boots8181
5 http://en.wikibooks.org/w/index.php?title=User:Breakpoint
6 http://en.wikibooks.org/w/index.php?title=User:Dallas1278
7 http://en.wikibooks.org/w/index.php?title=User:Damian_Yerrick
8 http://en.wikibooks.org/w/index.php?title=User:Darklama
9 http://en.wikibooks.org/w/index.php?title=User:DavidCary
10 http://en.wikibooks.org/w/index.php?title=User:DavidL
11 http://en.wikibooks.org/w/index.php?title=User:Derbeth
12 http://en.wikibooks.org/w/index.php?title=User:Dirk_H%C3%BCnniger
13 http://en.wikibooks.org/w/index.php?title=User:EdDavies
14 http://en.wikibooks.org/w/index.php?title=User:Fishpi
15 http://en.wikibooks.org/w/index.php?title=User:Geocachernemesis
16 http://en.wikibooks.org/w/index.php?title=User:Guanabot
17 http://en.wikibooks.org/w/index.php?title=User:Gumba_gumba
18 http://en.wikibooks.org/w/index.php?title=User:HumbertoDiogenes
19 http://en.wikibooks.org/w/index.php?title=User:Insaneinside
20 http://en.wikibooks.org/w/index.php?title=User:JenVan
21 http://en.wikibooks.org/w/index.php?title=User:Jguk

3	Jhdiii[22]
4	Jomegat[23]
11	Lehoaithanh[24]
1	Micha s[25]
2	Mike518[26]
6	Netch[27]
1	Ninly[28]
4	Panic2k4[29]
15	QuiteUnusual[30]
6	Recent Runes[31]
80	Renffeh[32]
1	Rmallins[33]
89	Robert Horning[34]
1	Rustamabd[35]
1	Sandcat01[36]
1	Theodore.cackowski[37]
1	Trainsonplanes[38]
2	Wajidstar[39]
3	Webaware[40]
1	Xania[41]
3	Xenodevil[42]
1	Yuriybrisk[43]

22 http://en.wikibooks.org/w/index.php?title=User:Jhdiii
23 http://en.wikibooks.org/w/index.php?title=User:Jomegat
24 http://en.wikibooks.org/w/index.php?title=User:Lehoaithanh
25 http://en.wikibooks.org/w/index.php?title=User:Micha_s
26 http://en.wikibooks.org/w/index.php?title=User:Mike518
27 http://en.wikibooks.org/w/index.php?title=User:Netch
28 http://en.wikibooks.org/w/index.php?title=User:Ninly
29 http://en.wikibooks.org/w/index.php?title=User:Panic2k4
30 http://en.wikibooks.org/w/index.php?title=User:QuiteUnusual
31 http://en.wikibooks.org/w/index.php?title=User:Recent_Runes
32 http://en.wikibooks.org/w/index.php?title=User:Renffeh
33 http://en.wikibooks.org/w/index.php?title=User:Rmallins
34 http://en.wikibooks.org/w/index.php?title=User:Robert_Horning
35 http://en.wikibooks.org/w/index.php?title=User:Rustamabd
36 http://en.wikibooks.org/w/index.php?title=User:Sandcat01
37 http://en.wikibooks.org/w/index.php?title=User:Theodore.cackowski
38 http://en.wikibooks.org/w/index.php?title=User:Trainsonplanes
39 http://en.wikibooks.org/w/index.php?title=User:Wajidstar
40 http://en.wikibooks.org/w/index.php?title=User:Webaware
41 http://en.wikibooks.org/w/index.php?title=User:Xania
42 http://en.wikibooks.org/w/index.php?title=User:Xenodevil
43 http://en.wikibooks.org/w/index.php?title=User:Yuriybrisk

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. http://www.eclipse.org/org/documents/epl-v10.php

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[44]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

44 Chapter 11 on page 145

1		GFDL
2	User Mike1024[45]	PD
3	Afrank99[46]	GFDL
4	User Smial[47] on de.wikipedia[48]	cc-by-sa-2.0
5	Afrank99[49]	cc-by-sa-2.5
6		GFDL

45 http://en.wikibooks.org/wiki/User%3AMike1024
46 http://en.wikibooks.org/wiki/User%3AAfrank99
47 http://en.wikibooks.org/wiki/%3Ade%3ABenutzer%3ASmial
48 http://de.wikipedia.org
49 http://en.wikibooks.org/wiki/User%3AAfrank99

11 Licenses

11.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work)

from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

11.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title

Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

11.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.