*** The `Linux Serial Port Programming mini-HOWTO' is posted automatically by
the
*** Linux HOWTO coordinator, Greg Hankins <gregh@sunsite.unc.edu>. Please
*** direct any comments or questions about this HOWTO to the author,
*** Antonino Iannella <antonino@usa.net>.

- --- BEGIN Linux Serial Port Programming mini-HOWTO part 1/1 ---

Linux Serial Port Programming Mini-Howto
Antonino Iannella, antonino@usa.net
Version 1.0, March 9th 1997

1. Introduction

This document describes how to program the RS-232 ports for serial communications, under PC-Linux. It covers information about the serial ports, RS232 connections, modem issues, and the C programming logic.

2. Background

For our final year project, our group had to design a concept World Wide Web browser. Our prototype was a hand-held device which plugged into a PC's COM2 (25-pin) serial port. The user would issue commands to the browser (eg Back, Open, etc) by sending character commands to the PC. The browser software would detect it, and perform the required operation.

The method provided in the 'Linux I/O port programming mini-HOWTO' did not act reliably. Often, an incorrect value would be received. The information within provided a 100% reliable, quasi-POSIX-compliant approach to communication.

The program provided at the end of this 0 formed the basis of the PC's communication program. This document is written from the 1 needed for the web browser project. It revolves around C programming for Linux, for a 9600 baud device attached to COM2.

3. Acknowledgements

The information here comes mainly from these sources -

Heavily based on the 'Serial Programming Guide for POSIX Compliant Operating Systems', at http://www.easysw.com/~mike/serial/, by mike@easysw.com. If you need greater detail or more information, then this is the place to visit. Most of the information in this 0 originated here.

The 'Linux I/O port programming mini-HOWTO', by Riku Saikkonen (rjs@spider.compart.fi). This document provides a different approach to Linux serial programming.

'The Linux Serial HOWTO', by Greg Hankins, greg.hankins@cc.gatech.edu describes how to set up serial communications devices on a Linux box. It describes many aspects of serial devices and 0.

My two wonderful project partners: Gerel Enrile and Joyce Gong.

4. Copyright

It is intended for the general public. it may be reproduced and distributed in whole or in part, using any electronic or physical medium. However, this copyright notice must remain on all copies.

Please forward question, suggestions, corrections, and all tidbits of information to the author at antonino@usa.net
(or nettuno@light.iinet.net.au).
You will be acknowledged in future HOWTO revisions.

5. RS-232 connections

RS-232 is a standard for serial communications. It comes in different varieties. The most common is RS-232C which defines a 'mark' bit as a voltage between -3V and -12V, and a 'space' bit as a voltage between +3V and +12V. RS-574 is the standard for 9-pin PC connectors and voltages.

RS-232 basically consists of wires for serial communications; sending and receiving, timing, status, and handshaking.
You can use a null modem cable as your connector.
The following pins are what were used for our project. We connected the device to the DB-25 pin COM2 port. Please note that the Transmit line from the PC must connect to the Receive line of the device, and vice versa. Also, please note that a parallel port is different to a serial port!

		PC's pins	Device's pins		
TxD	Transmit Data	2 - 3		RxD	Receive Data
RxD	Receive Data	3 - 2		TxD	Transmit Data
SG	Signal Ground	7 - 7		SG	Signal Ground

Refer to the Linux Serial HOWTO for more specialised connections, and detailed RS-232 pins.

6. Serial 0 and RS-232 definitions

The way that data get transmitted in serial communications is, well, serially. One data bit is sent at a time. Each bit is either on (or the 'mark' state), or off (or the 'space' state).

The serial data throughput is usually expressed in bits-per-second (bps) or baudot (baud). Throughput is the number of data bits (2 or off) that may be sent in a second. Your modem might be able to support 115200 baud. The project's web browser device was designed to run at 9600 baud.

RS-232 provides 18 different signals. About 6 are available to UNIX for programming.

GND - Logic ground

Very important. Acts as a reference voltage, so the devices know the relative voltage of the data transmitted.

TxD - Transmitted data

Carries the data transmitted from your PC.
A 'mark' voltage is interpreted as 1,
while a 'space' voltage is interpreted as 0.

RxD - Received data

Carries the data transmitted to your PC from the other device.

DCD - Data carrier detect

Is sent from the other device to your PC. A 'space' voltage means that the device is still connected, or 'on-line'. This signal is not always used or available.

DTR - Data terminal ready

Is sent from your PC to tell the device that you are ready (space) or not-ready (mark). DTR is usually enabled automatically whenever you access the serial interface.

CTS - Clear to send

Is sent from the other device to your PC.
A 'space' voltage means that your PC may send some data.
It is usually used to regulate the flow of serial data from your PC, but it is not currently supported by all UNIX flavours.

RTS - Request to send

Is set to the 'space' voltage by your PC when it requests to send more data. It also used to regulate data flow. Many systems leave it on 'space' voltage all the time.

7. Communication issues

This section covers other issues of serial communication which might be relevant to your particular application.
Since we are programming for asynchronous communication, we need the PCs/devices to know where each character starts and ends in the serial data flow.

In asynchronous mode, the serial data line stays in the mark state until a character is sent. A 'start bit' is sent before each character; it is always 0 and tells the PC/device that a character will follow.
After the start bit, the character's bits are sent, then a 'parity' bit, and one or more stop bits.

The parity bit is a checksum of the data bits, indicating the number of 1 bits it contained -

```
Even parity  - parity bit is 1 if there is an even number of 1s
Odd parity   - parity bit is 1 if there is an odd number of 1s
Space parity - parity bit is always 0
Mark parity  - parity bit is 1
No parity    - no parity bit is sent or present.
```

'Stop' bits come at the end of every character. There may be 1, 1.5, or 2 stop bits. They used to be used to give the computer time to process the character, but now they are used to synchronise the computer to the incoming characters.

Asynchronous data is usually described like '8N1' - 8 data bits, no parity bits, 1 stop bit. Another common one is '7N1'.

Flow control is used to regulate the data flow between devices, if there is some sort of limitation, such as a slow device. There is 'Software Flow Control' using special characters, XON and XOFF, to regulate the flow. This method is not useful for transmitting non-textual data.
'Hardware Flow Control' uses the RTS and CTS signals instead of special

characters. The receiver sets CTS to space when it is ready to receive, and to mark when it's not. The sender uses RTS the same way. This method is faster than Software Flow Control, since it uses a separate set of signals instead of extra bits.

Since the receive or transmit signal is at mark voltage until a new character is sent. A 'break' condition exists when the line is set to low for 1/4 to 1/2 a second. It is used to reset a communications line, or change the operation mode of devices like modems.

8. Basic port programming

Hopefully, all of the above is reasonably clear to you, so you may proceed to program with confidence!

In UNIX all system devices are treated as (special) files. All serial ports are opened, read from, written to, and closed, just like a binary file. In Linux, the PC serial ports are

```
   COM1 - /dev/ttyS0
   COM2 - /dev/ttyS1
   COM3 - /dev/ttyS2
   COM4 - /dev/ttyS3
```

Firstly, open the serial port as a file. However, UNIX does not allow devices to be accessed by normal users. To solve this, either run the program as the superuser, or change the permission on the device as root, eg

```
   chmod a+rw /dev/ttyS1          (lets everyone access COM2)
```

To open the file do the following. Notice the three flags used in the open() function. O_RDWR means that we open the port for reading and writing. O_NOCTTY specifies that the program won't be the controlling entity for the port. Most user programs don't want this feature. O_NDELAY means that your program ignores the DCD line. If it didn't, the program will be put to sleep until DCD is set to 'space' voltage.

```
/*
* 'open_port()' - Open serial port 1 - COM2.
*
* Returns the file descriptor on success or -1 on error.
*/

int open_port(void)
{
  int fd;                                     /* File descriptor for the port */

  fd = open(&amp;quot;/dev/ttyS1&amp;quot;, O_RDWR | O_NOCTTY | O_NDELAY);
  if (fd == -1)
  {                                                /* Could not open the port */
    fprintf(stderr, &amp;quot;open_port: Unable to open /dev/ttyS1 - %s\n&amp;quot;,
            strerror(errno));
  }

  return (fd);
}
```

If you need to write data to the port, do something like

```
  n = write(fd, &amp;quot;ATZ\r&amp;quot;, 4);

  if (n &amp;lt; 0)
```

```
     puts(&quot;write() of 4 bytes failed!\n&quot;);
```

Reading from the port is more complicated. If you open the port in 'raw data' mode (the norm), each read() returns the number of characters actually available in the serial buffers. However, if no characters are available, read() will block until it receives characters, an interval timer expires, or an error occurs. Use the following to make read return immediately. FNDELAY makes read() return 0 if no characters were read.

```
  fcntl(mainfd, F_SETFL, FNDELAY);             /* Configure port reading */
```

To close the serial port, simply use

```
  close(fd);
```

9. Port configuration

This section discusses how to configure the serial port for your device. You will need to set the terminal attributes related to the port. To do this, include <termios.h> and access the termios structure using the POSIX tcgetattr() and tcsetattr() functions.

The termios structure contains

```
  c_cflag - Control options
  c_lflag - Line options
  c_iflag - Input options
  c_oflag - Output options
  c_cc    - Control characters
```

See section 12 for a list of c_cflag control modes. They are used to set the baud rate, parity and stop bits, and flow control. Always enable CLOCAL and CREAD, so the program does not own the port, and so the serial interface driver will read incoming bytes.

9.1. Accessing the termios structure and the baud rate

Use cfsetospeed() and cfsetispeed() to set the baud rate. CRTSCTS might be called CNEW_RTSCTS on other systems. The following uses a termios structure called 'options'. For our project, the device transmitted at 9600 baud and transmitted nothing special.

```
 tcgetattr(mainfd, &amp;options);        /* Get the current options for the port */
 cfsetispeed(&amp;options, B9600);                /* Set the baud rates to 9600 */
 cfsetospeed(&amp;options, B9600);
                                    /* Enable the receiver and set local mode */
 options.c_cflag |= (CLOCAL | CREAD);

                                          /* Set the new options for the port */
 tcsetattr(mainfd, TCSANOW, &amp;options);
```

The tcsetattr() function replaces the port's termios structure with the settings you provided. The TCSANOW constant means that the changes should occur immediately, without waiting for data transmission to complete. Alternative choices are TCSADRAIN and TCSAFLUSH, which wait until buffers are cleared. Refer to section 13.

9.2. Character size and parity

To set the character size, you must use bitwise logic.
The following code sets the character size to 8 data bits, and no parity.

```
   options.c_cflag &amp;= ~PARENB;  /* Mask character size to 8 bits, no parity
*/
   options.c_cflag &amp;= ~CSTOPB;
   options.c_cflag &amp;= ~CSIZE;
   options.c_cflag |=  CS8;                          /* Select 8 data bits */
```

For other methods, refer to section 11.

9.3. Hardware flow control

To enable hardware flow control, use

```
    options.c_cflag |= CRTSCTS;            /* Enable hardware flow control */
```

To disable it,

```
    options.c_cflag &amp;= ~CRTSCTS;          /* Disable hardware flow control
*/
```

9.4. Canonical and raw input

Canonical input means that all incoming characters are placed in a buffer which may be edited by the user, until a carriage return or line feed (CR or LF) are received. Typically, you would use

```
 options.c_lflag |= ~(ICANON | ECHO | ECHOE);
```

Raw input is unprocessed, so they may be used as they are read.
Our device sent raw data.

```
 options.c_lflag &amp;= ~(ICANON | ECHO | ISIG);
```

Whether you use canonical or raw input, make sure you never enable input echo when connected to a computer/device which is echoing characters to you.
Refer to section 14 for local mode constants.

9.5. POSIX input modes

Set the port's input modes for any input processing.
Set input parity when you have enabled parity in the c_cflag part.
Usually you'd use the following to enable parity checking, and strip the parity bit off the data, before your program reads it.

```
 options.c_iflag |= (INPCK | ISTRIP);
```

You might use IGNPAR, which ignores all parity errors. PARMRK marks parity errors by inserting a DEL(255) and NUL character before the bad character.
If IGNPAR is enabled, only a NUL is inserted.

You may set software flow control using

```
 options.c_iflag |= (IXON | IXOFF | IXANY);
```

Refer to section 15 for input mode constants.

9.6. POSIX output modes

To set port output modes, use the c_oflag member.
To select processed output, use the following. Of all the output modes, you will probably only use ONCLR to convert newlines into CR and LFs.

```
 options.c_oflag |= OPOST;
```

For raw output, use

```
 options.c_oflag &amp;= ~OPOST;
```

Refer to section 16 for output mode constants.

9.7. POSIX control modes

You may set the control characters using the c_cc part.
Set the software flow control characters in the VSTART and VSTOP
elements. The standard is DC1(17) and DC3(19) for XON and XOFF.
VMIN specifies the minimum number of 0 to read. If it is 0, then
VTIME specifies the time to wait for each character.
If VMIN is not 0, VTIME is the time to wait to read the first character.
If the first character is read, then any read() will be blocked until all
VMIN characters are read.
VTIME is specified in tenths of seconds. If it is 0, then read()s will
be permanently blocked unless NDELAY was previously specified with fcntl().

Refer to section 17 for control mode constants.

10. Sample program

This program is a skeleton COM2 reader, which was used for our project.
It did not need all of the information specified above for
configuring ports. The 20ms delay is used to indicate that data coming into
the port is buffered, and is available for the next read().

```
/* Better port reading program
  v1.0
  23-10-96

  This test program uses quasi-POSIX compliant UNIX functions to
  open the ABU port and read.

  Uses termio functions to initialise the port to 9600 baud, at
  8 data bits, no parity, no hardware flow control,
  and features character buffering.
  The 20ms delay after the port read indicates that characters are
  buffered if a button is pressed many times.

  This program was derived from instructions at
   http://www.easysw.com/~mike/serial/
*/

#include &lt;stdio.h&gt;   /* Standard input/output definitions */
#include &lt;string.h&gt;  /* String function definitions */
#include &lt;unistd.h&gt;  /* UNIX standard function definitions */
#include &lt;fcntl.h&gt;   /* File control definitions */
#include &lt;errno.h&gt;   /* Error number definitions */
#include &lt;termios.h&gt; /* POSIX terminal control definitions */

/*
 * 'open_port()' - Open serial port 1.
 *
 * Returns the file descriptor on success or -1 on error.
 */

int open_port(void)
{
```

```
   int fd;                                    /* File descriptor for the port */

   fd = open(&quot;/dev/ttyS1&quot;, O_RDWR | O_NOCTTY | O_NDELAY);

   if (fd == -1)
   {                                                /* Could not open the port */
     fprintf(stderr, &quot;open_port: Unable to open /dev/ttyS1 - %s\n&quot;,
             strerror(errno));
   }

   return (fd);
 }

void main()
{
 int mainfd=0;                                          /* File descriptor */
 char chout;
 struct termios options;

 mainfd = open_port();

 fcntl(mainfd, F_SETFL, FNDELAY);                  /* Configure port reading */
                                  /* Get the current options for the port */
 tcgetattr(mainfd, &amp;options);
 cfsetispeed(&amp;options, B9600);                 /* Set the baud rates to 9600
*/
 cfsetospeed(&amp;options, B9600);

                                    /* Enable the receiver and set local mode */
 options.c_cflag |= (CLOCAL | CREAD);
 options.c_cflag &amp;= ~PARENB; /* Mask the character size to 8 bits, no parity
*/
 options.c_cflag &amp;= ~CSTOPB;
 options.c_cflag &amp;= ~CSIZE;
 options.c_cflag |=  CS8;                               /* Select 8 data bits */
 options.c_cflag &amp;= ~CRTSCTS;               /* Disable hardware flow control
*/

                                  /* Enable data to be processed as raw input */
 options.c_lflag &amp;= ~(ICANON | ECHO | ISIG);

                                       /* Set the new options for the port */
 tcsetattr(mainfd, TCSANOW, &amp;options);

 while (1)
 {
   read(mainfd, &amp;chout, sizeof(chout));          /* Read character from ABU
*/

   if (chout != 0)
      printf(&quot;Got %c.\n&quot;, chout);

   chout=0;
   usleep(20000);
 }
                                                    /* Close the serial port */
  close(mainfd);
 }
```

11. Character and parity settings

No parity (8N1):

```
 options.c_cflag &amp;= ~PARENB;
 options.c_cflag &amp;= ~CSTOPB;
 options.c_cflag &amp;= ~CSIZE;
 options.c_cflag |= CS8;
```

Even parity (7E1):

```
 options.c_cflag |= PARENB;
 options.c_cflag &amp;= ~PARODD;
 options.c_cflag &amp;= ~CSTOPB;
 options.c_cflag &amp;= ~CSIZE;
 options.c_cflag |= CS7;
```

Odd parity (7O1):

```
 options.c_cflag |= PARENB;
 options.c_cflag |= PARODD;
 options.c_cflag &amp;= ~CSTOPB;
 options.c_cflag &amp;= ~CSIZE;
 options.c_cflag |= CS7;
```

Mark parity is simulated by using 2 stop bits (7M1):

```
 options.c_cflag &amp;= ~PARENB;
 options.c_cflag |= CSTOPB;
 options.c_cflag &amp;= ~CSIZE;
 options.c_cflag |= CS7;
```

Space parity is setup the same as no parity (7S1):

```
 options.c_cflag &amp;= ~PARENB;
 options.c_cflag &amp;= ~CSTOPB;
 options.c_cflag &amp;= ~CSIZE;
 options.c_cflag |= CS8;
```

12. POSIX control mode flags

The following table lists the possible control modes for c_cflag.

Constant	Description
CBAUD	Bit mask for baud rate
B0	0 baud (drop DTR)
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
EXTA	External rate clock
EXTB	External rate clock
CSIZE	Bit mask for data bits
CS5	5 data bits

CS6	6 data bits
CS7	7 data bits
CS8	8 data bits
CSTOPB	2 stop bits (1 otherwise)
CREAD	Enable receiver
PARENB	Enable parity bit
PARODD	Use odd parity instead of even
HUPCL	Hangup (drop DTR) on last close
CLOCAL	Local line - do not change 'owner' of port
LOBLK	Block job control output
CRTSCTS	Enable hardware flow control (not supported on all platforms)

13. POSIX tcsetattr Constants

Constant	Description
TCSANOW	Make changes now without waiting for data to complete
TCSADRAIN	Wait until everything has been transmitted
TCSAFLUSH	Flush input and output buffers and make the change

14. POSIX Local Mode Constants

Constant	Description
ISIG	Enable SIGINTR, SIGSUSP, SIGDSUSP, and SIGQUIT signals
ICANON	Enable canonical input (else raw)
XCASE	Map uppercase \lowercase (obselete)
ECHO	Enable echoing of input characters
ECHOE	Echo erase character as BS-SP-BS
ECHOK	Echo NL after kill character
ECHONL	Echo NL
NOFLSH	Disable flushing of input buffers after interrupt or quit characters
IEXTEN	Enable extended functions
ECHOCTL	Echo control characters as ^char and delete as ~?
ECHOPRT	Echo erased character as character erased
ECHOKE	BS-SP-BS entire line on line kill
FLUSHO	Output being flushed
PENDIN	Retype pending input at next read or input char
TOSTOP	Send SIGTTOU for background output

15. POSIX Input Mode Constants

Constant	Description
INPCK	Enable parity check
IGNPAR	Ignore parity errors
PARMRK	Mark parity errors
ISTRIP	Strip parity bits
IXON	Enable software flow control (outgoing)
IXOFF	Enable software flow control (incoming)
IXANY	Allow any character to start flow again
IGNBRK	Ignore break condition
BRKINT	Send a SIGINT when a break condition is detected
INLCR	Map NL to CR
IGNCR	Ignore CR
ICRNL	Map CR to NL
IUCLC	Map uppercase to lowercase
IMAXBEL	Echo BEL on input line too long

16. POSIX Output Mode Constants

Constant	Description
OPOST	Postprocess output (not set = raw output)
OLCUC	Map lowercase to uppercase
ONLCR	Map NL to CR-NL
OCRNL	Map CR to NL
NOCR	No CR output at column 0
ONLRET	NL performs CR function
OFILL	Use fill characters for delay
OFDEL	Fill character is DEL
NLDLY	Mask for delay time needed between lines
NL0	No delay for NLs
NL1	Delay further output after newline for 100 milliseconds
CRDLY	Mask for delay time needed to return carriage to left column
CR0	No delay for CRs
CR1	Delay after CRs depending on current column position
CR2	Delay 100 milliseconds after sending CRs
CR3	Delay 150 milliseconds after sending CRs
TABDLY	Mask for delay time needed after TABs
TAB0	No delay for TABs
TAB1	Delay after TABs depending on current column position
TAB2	Delay 100 milliseconds after sending TABs
TAB3	Expand TAB characters to spaces
BSDLY	Mask for delay time needed after BSs
BS0	No delay for BSs
BS1	Delay 50 milliseconds after sending BSs
VTDLY	Mask for delay time needed after VTs
VT0	No delay for VTs
VT1	Delay 2 seconds after sending VTs
FFDLY	Mask for delay time needed after FFs
FF0	No delay for FFs
FF1	Delay 2 seconds after sending FFs

17. POSIX Control Character Constants

Constant	Description	Key
VINTR	Interrupt	CTRL-C
VQUIT	Quit	CTRL-Z
VERASE	Erase	Backspace (BS)
VKILL	Kill-line	CTRL-U
VEOF	End-of-file	CTRL-D
VEOL	End-of-line	Carriage return (CR)
VEOL2	Second end-of-line	Line feed (LF)
VMIN	Minimum number of characters to read	
VTIME	Time to wait for data (tenths of seconds)	

---------- End of Linux Serial Programming Mini-Howto ----------