Serial Programming HOWTO

Gary Frerking

gary@frerking.org

Peter Baumann

Revision History

Revision 1.01	2001−08−26	Revised by: glf
New maintainer, converted to DocBook		
Revision 1.0	1998−01−22	Revised by: phb
Initial document release		

This document describes how to program communications with devices over a serial port on a Linux box.

Table of Contents

1. Introduction.....1
1.1. Copyright Information.....1
1.2. Disclaimer.....1
1.3. New Versions.....2
1.4. Credits.....2
1.5. Feedback.....2

2. Getting started.....3
2.1. Debugging.....3
2.2. Port Settings.....3
2.3. Input Concepts for Serial Devices.....4
2.3.1. Canonical Input Processing.....4
2.3.2. Non−Canonical Input Processing.....4
2.3.3. Asynchronous Input.....4
2.3.4. Waiting for Input from Multiple Sources.....4

3. Program Examples.....6
3.1. Canonical Input Processing.....6
3.2. Non−Canonical Input Processing.....8
3.3. Asynchronous Input.....9
3.4. Waiting for Input from Multiple Sources.....11

4. Other Sources of Information.....13

1. Introduction

This is the Linux Serial Programming HOWTO. All about how to program communications with other devices / computers over a serial line under Linux. Different techniques are explained: Canonical I/O (only complete lines are transmitted/received), asyncronous I/O, and waiting for input from multiple sources.

This is the first update to the initial release of the Linux Serial Programming HOWTO. The primary purpose of this update is to change the author information and convert the document to DocBook format. In terms of technical content, very little if anything has changed at this time. Sweeping changes to the technical content aren't going to happen overnight, but I'll work on it as much as time allows.

If you've been waiting in the wings for someone to take over this HOWTO, you've gotten your wish. Please send me any and all feedback you have, it'd be very much appreciated.

All examples were tested using a i386 Linux Kernel 2.0.29.

1.1. Copyright Information

1.2. Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your system. Proceed with caution, and although this is highly unlikely, the author(s) do not take any responsibility for that.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

You are strongly recommended to take a backup of your system before major installation and backups at regular intervals.

1.3. New Versions

As previously mentioned, not much is new in terms of technical content yet.

1.4. Credits

The original author thanked Mr. Strudthoff, Michael Carter, Peter Waltenberg, Antonino Ianella, Greg Hankins, Dave Pfaltzgraff, Sean Lincolne, Michael Wiedmann, and Adrey Bonar.

1.5. Feedback

Feedback is most certainly welcome for this document. Without your submissions and input, this document wouldn't exist. Please send your additions, comments and criticisms to the following email address : `<gary@frerking.org>`.

2. Getting started

2.1. Debugging

The best way to debug your code is to set up another Linux box, and connect the two computers via a null−modem cable. Use miniterm (available from the LDP programmers guide (`ftp://sunsite.unc.edu/pub/Linux/docs/LDP/programmers-guide/lpg-0.4.tar.gz` in the examples directory) to transmit characters to your Linux box. Miniterm can be compiled very easily and will transmit all keyboard input raw over the serial port. Only the define statement `#define MODEMDEVICE "/dev/ttyS0"` has to be checked. Set it to `ttyS0` for COM1, `ttyS1` for COM2, etc.. It is essential for testing, that *all* characters are transmitted raw (without output processing) over the line. To test your connection, start miniterm on both computers and just type away. The characters input on one computer should appear on the other computer and vice versa. The input will not be echoed to the attached screen.

To make a null−modem cable you have to cross the TxD (transmit) and RxD (receive) lines. For a description of a cable see sect. 7 of the Serial−HOWTO.

It is also possible to perform this testing with only one computer, if you have two unused serial ports. You can then run two miniterms off two virtual consoles. If you free a serial port by disconnecting the mouse, remember to redirect `/dev/mouse` if it exists. If you use a multiport serial card, be sure to configure it correctly. I had mine configured wrong and everything worked fine as long as I was testing only on my computer. When I connected to another computer, the port started loosing characters. Executing two programs on one computer just isn't fully asynchronous.

2.2. Port Settings

The devices `/dev/ttyS*` are intended to hook up terminals to your Linux box, and are configured for this use after startup. This has to be kept in mind when programming communication with a raw device. E.g. the ports are configured to echo characters sent from the device back to it, which normally has to be changed for data transmission.

All parameters can be easily configured from within a program. The configuration is stored in a structure `struct termios`, which is defined in `<asm/termbits.h>`:

```
        #define NCCS 19
        struct termios {
                tcflag_t c_iflag;               /* input mode flags */
                tcflag_t c_oflag;               /* output mode flags */
                tcflag_t c_cflag;               /* control mode flags */
                tcflag_t c_lflag;               /* local mode flags */
                cc_t c_line;                    /* line discipline */
                cc_t c_cc[NCCS];                /* control characters */
        };
```

This file also includes all flag definitions. The input mode flags in `c_iflag` handle all input processing, which means that the characters sent from the device can be processed before they are read with `read`. Similarly `c_oflag` handles the output processing. `c_cflag` contains the settings for the port, as the baudrate, bits per character, stop bits, etc.. The local mode flags stored in `c_lflag` determine if characters are echoed, signals are sent to your program, etc.. Finally the array `c_cc` defines the control characters for end of file, stop, etc.. Default values for the control characters are defined in `<asm/termios.h>`. The flags are described in the manual page `termios(3)`. The structure `termios` contains the `c_line` (line discipline) element, which is not used in POSIX compliant systems.

2.3. Input Concepts for Serial Devices

Here three different input concepts will be presented. The appropriate concept has to be chosen for the intended application. Whenever possible, do not loop reading single characters to get a complete string. When I did this, I lost characters, whereas a `read` for the whole string did not show any errors.

2.3.1. Canonical Input Processing

This is the normal processing mode for terminals, but can also be useful for communicating with other dl input is processed in units of lines, which means that a `read` will only return a full line of input. A line is by default terminated by a `NL` (ASCII `LF`), an end of file, or an end of line character. A `CR` (the DOS/Windows default end−of−line) will not terminate a line with the default settings.

Canonical input processing can also handle the erase, delete word, and reprint characters, translate `CR` to `NL`, etc..

2.3.2. Non−Canonical Input Processing

Non−Canonical Input Processing will handle a fixed amount of characters per read, and allows for a character timer. This mode should be used if your application will always read a fixed number of characters, or if the connected device sends bursts of characters.

2.3.3. Asynchronous Input

The two modes described above can be used in synchronous and asynchronous mode. Synchronous is the default, where a `read` statement will block, until the read is satisfied. In asynchronous mode the `read` statement will return immediatly and send a signal to the calling program upon completion. This signal can be received by a signal handler.

2.3.4. Waiting for Input from Multiple Sources

This is not a different input mode, but might be useful, if you are handling multiple devices. In my application I was handling input over a TCP/IP socket and input over a serial connection from another computer quasi−simultaneously. The program example given below will wait for input from two different input sources. If input from one source becomes available, it will be processed, and the program will then wait for new input.

The approach presented below seems rather complex, but it is important to keep in mind that Linux is a multi−processing operating system. The `select` system call will not load the CPU while waiting for input, whereas looping until input becomes available would slow down other processes executing at the same time.

---

3. Program Examples

All examples have been derived from `miniterm.c`. The type ahead buffer is limited to 255 characters, just like the maximum string length for canonical input processing (`<linux/limits.h>` or `<posix1_lim.h>`).

See the comments in the code for explanation of the use of the different input modes. I hope that the code is understandable. The example for canonical input is commented best, the other examples are commented only where they differ from the example for canonical input to emphasize the differences.

The descriptions are not complete, but you are encouraged to experiment with the examples to derive the best solution for your application.

Don't forget to give the appropriate serial ports the right permissions (e. g.: `chmod a+rw /dev/ttyS1`)!

3.1. Canonical Input Processing

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

/* baudrate settings are defined in <asm/termbits.h>, which is
included by <termios.h> */
#define BAUDRATE B38400
/* change this definition for the correct port */
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
  int fd,c, res;
  struct termios oldtio,newtio;
  char buf[255];
/*
  Open modem device for reading and writing and not as controlling tty
  because we don't want to get killed if linenoise sends CTRL-C.
*/
 fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
 if (fd <0) {perror(MODEMDEVICE); exit(-1); }

 tcgetattr(fd,&oldtio); /* save current serial port settings */
 bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */

/*
  BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
  CRTSCTS : output hardware flow control (only used if the cable has
            all necessary lines. See sect. 7 of Serial-HOWTO)
  CS8     : 8n1 (8bit,no parity,1 stopbit)
```

```
  CLOCAL  : local connection, no modem contol
  CREAD   : enable receiving characters
*/
 newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;

/*
  IGNPAR  : ignore bytes with parity errors
  ICRNL   : map CR to NL (otherwise a CR input on the other computer
            will not terminate input)
  otherwise make device raw (no other input processing)
*/
 newtio.c_iflag = IGNPAR | ICRNL;

/*
 Raw output.
*/
 newtio.c_oflag = 0;

/*
  ICANON  : enable canonical input
  disable all echo functionality, and don't send signals to calling program
*/
 newtio.c_lflag = ICANON;

/*
  initialize all control characters
  default values can be found in /usr/include/termios.h, and are given
  in the comments, but we don't need them here
*/
 newtio.c_cc[VINTR]    = 0;     /* Ctrl-c */
 newtio.c_cc[VQUIT]    = 0;     /* Ctrl-\ */
 newtio.c_cc[VERASE]   = 0;     /* del */
 newtio.c_cc[VKILL]    = 0;     /* @ */
 newtio.c_cc[VEOF]     = 4;     /* Ctrl-d */
 newtio.c_cc[VTIME]    = 0;     /* inter-character timer unused */
 newtio.c_cc[VMIN]     = 1;     /* blocking read until 1 character arrives */
 newtio.c_cc[VSWTC]    = 0;     /* '\0' */
 newtio.c_cc[VSTART]   = 0;     /* Ctrl-q */
 newtio.c_cc[VSTOP]    = 0;     /* Ctrl-s */
 newtio.c_cc[VSUSP]    = 0;     /* Ctrl-z */
 newtio.c_cc[VEOL]     = 0;     /* '\0' */
 newtio.c_cc[VREPRINT] = 0;     /* Ctrl-r */
 newtio.c_cc[VDISCARD] = 0;     /* Ctrl-u */
 newtio.c_cc[VWERASE]  = 0;     /* Ctrl-w */
 newtio.c_cc[VLNEXT]   = 0;     /* Ctrl-v */
 newtio.c_cc[VEOL2]    = 0;     /* '\0' */

/*
  now clean the modem line and activate the settings for the port
*/
 tcflush(fd, TCIFLUSH);
 tcsetattr(fd,TCSANOW,&newtio);

/*
  terminal settings done, now handle input
  In this example, inputting a 'z' at the beginning of a line will
  exit the program.
*/
 while (STOP==FALSE) {     /* loop until we have a terminating condition */
 /* read blocks program execution until a line terminating character is
    input, even if more than 255 chars are input. If the number
    of characters read is smaller than the number of chars available,
```

```
          subsequent reads will return the remaining chars. res will be set
          to the actual number of characters actually read */
          res = read(fd,buf,255);
          buf[res]=0;             /* set end of string, so we can printf */
          printf(":%s:%d\n", buf, res);
          if (buf[0]=='z') STOP=TRUE;
       }
       /* restore the old port settings */
       tcsetattr(fd,TCSANOW,&oldtio);
      }
```

3.2. Non−Canonical Input Processing

In non−canonical input processing mode, input is not assembled into lines and input processing (erase, kill, delete, etc.) does not occur. Two parameters control the behavior of this mode: `c_cc[VTIME]` sets the character timer, and `c_cc[VMIN]` sets the minimum number of characters to receive before satisfying the read.

If MIN > 0 and TIME = 0, MIN sets the number of characters to receive before the read is satisfied. As TIME is zero, the timer is not used.

If MIN = 0 and TIME > 0, TIME serves as a timeout value. The read will be satisfied if a single character is read, or TIME is exceeded (t = TIME *0.1 s). If TIME is exceeded, no character will be returned.

If MIN > 0 and TIME > 0, TIME serves as an inter−character timer. The read will be satisfied if MIN characters are received, or the time between two characters exceeds TIME. The timer is restarted every time a character is received and only becomes active after the first character has been received.

If MIN = 0 and TIME = 0, read will be satisfied immediately. The number of characters currently available, or the number of characters requested will be returned. According to Antonino (see contributions), you could issue a `fcntl(fd, F_SETFL, FNDELAY);` before reading to get the same result.

By modifying `newtio.c_cc[VTIME]` and `newtio.c_cc[VMIN]` all modes described above can be tested.

```
      #include <sys/types.h>
      #include <sys/stat.h>
      #include <fcntl.h>
      #include <termios.h>
      #include <stdio.h>

      #define BAUDRATE B38400
      #define MODEMDEVICE "/dev/ttyS1"
      #define _POSIX_SOURCE 1 /* POSIX compliant source */
      #define FALSE 0
      #define TRUE 1

      volatile int STOP=FALSE;

      main()
      {
        int fd,c, res;
```

```
  struct termios oldtio,newtio;
  char buf[255];

  fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
  if (fd <0) {perror(MODEMDEVICE); exit(-1); }

  tcgetattr(fd,&oldtio); /* save current port settings */

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;

  /* set input mode (non-canonical, no echo,...) */
  newtio.c_lflag = 0;

  newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
  newtio.c_cc[VMIN]     = 5;   /* blocking read until 5 chars received */

  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);


  while (STOP==FALSE) {       /* loop for input */
    res = read(fd,buf,255);   /* returns after 5 chars have been input */
    buf[res]=0;               /* so we can printf... */
    printf(":%s:%d\n", buf, res);
    if (buf[0]=='z') STOP=TRUE;
  }
  tcsetattr(fd,TCSANOW,&oldtio);
}
```

3.3. Asynchronous Input

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/types.h>

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

void signal_handler_IO (int status);   /* definition of signal handler */
int wait_flag=TRUE;                    /* TRUE while no signal received */

main()
{
  int fd,c, res;
  struct termios oldtio,newtio;
  struct sigaction saio;           /* definition of signal action */
  char buf[255];
```

```
  /* open the device to be non-blocking (read will return immediatly) */
  fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
  if (fd <0) {perror(MODEMDEVICE); exit(-1); }

  /* install the signal handler before making the device asynchronous */
  saio.sa_handler = signal_handler_IO;
  saio.sa_mask = 0;
  saio.sa_flags = 0;
  saio.sa_restorer = NULL;
  sigaction(SIGIO,&saio,NULL);

  /* allow the process to receive SIGIO */
  fcntl(fd, F_SETOWN, getpid());
  /* Make the file descriptor asynchronous (the manual page says only
     O_APPEND and O_NONBLOCK, will work with F_SETFL...) */
  fcntl(fd, F_SETFL, FASYNC);

  tcgetattr(fd,&oldtio); /* save current port settings */
  /* set new port settings for canonical input processing */
  newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR | ICRNL;
  newtio.c_oflag = 0;
  newtio.c_lflag = ICANON;
  newtio.c_cc[VMIN]=1;
  newtio.c_cc[VTIME]=0;
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);

  /* loop while waiting for input. normally we would do something
     useful here */
  while (STOP==FALSE) {
    printf(".\n");usleep(100000);
    /* after receiving SIGIO, wait_flag = FALSE, input is available
       and can be read */
    if (wait_flag==FALSE) {
      res = read(fd,buf,255);
      buf[res]=0;
      printf(":%s:%d\n", buf, res);
      if (res==1) STOP=TRUE; /* stop loop if only a CR was input */
      wait_flag = TRUE;      /* wait for new input */
    }
  }
  /* restore old port settings */
  tcsetattr(fd,TCSANOW,&oldtio);
}

/***************************************************************************
* signal handler. sets wait_flag to FALSE, to indicate above loop that     *
* characters have been received.                                           *
***************************************************************************/

void signal_handler_IO (int status)
{
  printf("received SIGIO signal.\n");
  wait_flag = FALSE;
}
```

3.4. Waiting for Input from Multiple Sources

This section is kept to a minimum. It is just intended to be a hint, and therefore the example code is kept short. This will not only work with serial ports, but with any set of file descriptors.

The select call and accompanying macros use a `fd_set`. This is a bit array, which has a bit entry for every valid file descriptor number. `select` will accept a `fd_set` with the bits set for the relevant file descriptors and returns a `fd_set`, in which the bits for the file descriptors are set where input, output, or an exception occurred. All handling of `fd_set` is done with the provided macros. See also the manual page `select(2)`.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
  int    fd1, fd2;  /* input sources 1 and 2 */
  fd_set readfs;    /* file descriptor set */
  int    maxfd;     /* maximum file desciptor used */
  int    loop=1;    /* loop while TRUE */

  /* open_input_source opens a device, sets the port correctly, and
     returns a file descriptor */
  fd1 = open_input_source("/dev/ttyS1");   /* COM2 */
  if (fd1<0) exit(0);
  fd2 = open_input_source("/dev/ttyS2");   /* COM3 */
  if (fd2<0) exit(0);
  maxfd = MAX (fd1, fd2)+1;  /* maximum bit entry (fd) to test */

  /* loop for input */
  while (loop) {
    FD_SET(fd1, &readfs);  /* set testing for source 1 */
    FD_SET(fd2, &readfs);  /* set testing for source 2 */
    /* block until input becomes available */
    select(maxfd, &readfs, NULL, NULL, NULL);
    if (FD_ISSET(fd1))         /* input from source 1 available */
      handle_input_from_source1();
    if (FD_ISSET(fd2))         /* input from source 2 available */
      handle_input_from_source2();
  }
}
```

The given example blocks indefinitely, until input from one of the sources becomes available. If you need to timeout on input, just replace the select call by:

```
  int res;
  struct timeval Timeout;

  /* set timeout value within input loop */
  Timeout.tv_usec = 0;  /* milliseconds */
  Timeout.tv_sec  = 1;  /* seconds */
  res = select(maxfd, &readfs, NULL, NULL, &Timeout);
  if (res==0)
  /* number of file descriptors with input = 0, timeout occurred. */
```

This example will timeout after 1 second. If a timeout occurs, select will return 0, but beware that `Timeout` is decremented by the time actually waited for input by `select`. If the timeout value is zero, select will return immediatly.

---

4. Other Sources of Information

- The Linux Serial−HOWTO describes how to set up serial ports and contains hardware information.
- Serial Programming Guide for POSIX Compliant Operating Systems, by Michael Sweet.
- The manual page `termios(3)` describes all flags for the `termios` structure.