Linux serial port using C/C++

⇒ Everything in linux is file.
  ↳ Serial port is represented by files.
    { /dev/ttyUSB0 for most usb to serial cable }

⇒ To write to a serial port you write to a file and to read from a serial port, you read from the file.
  ↳ baud rate, parity etc is set by a special tty configuration structure.

★ **Basic setup in C** { Also applicable to C++ }

Studio.h { C library header }
String.h

fcntl.h
errno.h
termios.h
unistd.h
{ Linux Headers }

⇒ Opening serial port device:

```
int serial_port = open("/dev/ttyUSB0", O_RDWR);

# Check for errors
if (serial_port < 0) {
    printf("Error %i from open: %s\n", errno, strerror(errno));
}
```

flags
- O_RDONLY { Read only }
- O_WRONLY { Write only }
- O_CREAT { Create file if it dosent exist }
- O_EXCL { Prevent creation if it already exist }

errno = 2 { No such file or directory }
errno = 13 { Permission denied }
  ↳ This happens when current user is not part of the dialog group.

⇒ When modifying any configuration value, it is best Practice to only modify the bit you are interested in, and leave all other bits of the field untouched.

★ Configuration setup

⇒ We need to access termios struct in order to configure the serial port.

tcgetattr()
tcsetattr()

```
// Create new termios struc, we call it 'tty' for convention
struct termios tty;
memset(&tty, 0, sizeof tty);

// Read in existing settings, and handle any error
if(tcgetattr(serial_port, &tty) != 0) {
    printf("Error %i from tcgetattr: %s\n", errno, strerror(errno));
}
```

★ Control Modes (c_cflag)

⇒ The c_cflag member of the termios structure contains control parameter field.

5.1. PARENB (Parity)

If this bit is set, generation and detection of the parity bit is enabled. Most serial communications do not use a parity bit, so if you are unsure, clear this bit.

```
tty.c_cflag &= ~PARENB; // Clear parity bit, disabling parity (most common)
tty.c_cflag |= PARENB;  // Set parity bit, enabling parity
```

5.2. CSTOPB (Num. Stop Bits)

If this bit is set, two stop bits are used. If this is cleared, only one stop bit is used. Most serial communications only use one stop bit.

```
tty.c_cflag &= ~CSTOPB; // Clear stop field, only one stop bit used in communicatio
tty.c_cflag |= CSTOPB;  // Set stop field, two stop bits used in communication
```

→ {Most Common}

5.3. Number Of Bits Per Byte

The `CS<number>` fields set how many data bits are transmitted per byte across the serial port. The most common setting here is 8 (`CS8`). Definitely use this if you are unsure, I have never used a serial port before which didn't use 8 (but they do exist).

```
tty.c_cflag |= CS5; // 5 bits per byte
tty.c_cflag |= CS6; // 6 bits per byte
tty.c_cflag |= CS7; // 7 bits per byte
tty.c_cflag |= CS8; // 8 bits per byte (most common)
```

5.4. Flow Control (CRTSCTS)

If the `CRTSCTS` field is set, hardware RTS/CTS flow control is enabled. The most common setting here is to disable it. Enabling this when it should be disabled can result in your serial port receiving no data, as the sender will buffer it indefinitely, waiting for you to be "ready".

```
tty.c_cflag &= ~CRTSCTS; // Disable RTS/CTS hardware flow control (most common)
tty.c_cflag |= CRTSCTS;  // Enable RTS/CTS hardware flow control
```

5.5. CREAD and CLOCAL

Setting `CLOCAL` disables modem-specific signal lines such as carrier detect. Is also prevents the controlling process from getting sent a `SIGHUP` signal when a modem disconnect is detected, which is usually a good thing here. Setting `CLOCAL` allows us to read data (we definitely want that!).

```
tty.c_cflag |= CREAD | CLOCAL; // Turn on READ & ignore ctrl lines (CLOCAL = 1)
```

★ Local mode (c_lflag)

6.1. Disabling Canonical Mode

UNIX systems provide two basic modes of input, **canonical** and **non-canonical mode**. In canonical mode, input is processed when a new line character is received. The receiving application receives that data line-by-line. This is usually undesirable when dealing with a serial port, and so we normally want to disable canonical mode.

Canonical mode is disabled with:

```
tty.c_lflag &= ~ICANON;
```

Also, in canonical mode, some characters such as backspace are treated specially, and are used to edit the current line of text (erase). Again, we don't want this feature if processing raw serial data, as it will cause particular bytes to go missing!

6.2. Echo

If this bit is set, sent characters will be echoed back. Because we disabled canonical mode, I don't think these bits actually do anything, but it doesn't harm to disable them just in case!

```
tty.c_lflag &= ~ECHO; // Disable echo
tty.c_lflag &= ~ECHOE; // Disable erasure
tty.c_lflag &= ~ECHONL; // Disable new-line echo
```

6.3. Disable Signal Chars

When the `ISIG` bit is set, `INTR`, `QUIT` and `SUSP` characters are interpreted. We don't want this with a serial port, so clear this bit:

```
tty.c_lflag &= ~ISIG; // Disable interpretation of INTR, QUIT and SUSP
```

★ Input mode (c_iflag)

⇒ Low-level settings for input processing
↳ The c_iflag member is an int.

7.1. Software Flow Control (IXOFF, IXON, IXANY)

Clearing `IXOFF`, `IXON` and `IXANY` disables software flow control, which we don't want:

```
tty.c_iflag &= ~(IXON | IXOFF | IXANY); // Turn off s/w flow ctrl
```

7.2. Disabling Special Handling Of Bytes On Receive

Clearing all of the following bits disables any special handling of the bytes as they are received by the serial port, before they are passed to the application. We just want the raw data thanks!

```
tty.c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL); // Disable any spe
```

★ Output Modes (c_oflag)

⇒ Low level setting for output processing.

⇒ We want to disable any special handling of output char/bytes

```
tty.c_oflag &= ~OPOST; // Prevent special interpretation of output bytes (e.g. newline chars)
tty.c_oflag &= ~ONLCR; // Prevent conversion of newline to carriage return/line feed
```

★ VMIN and VTIME (c_cc)

VMIN = 0, VTIME = 0: No blocking, return immediately with what is available

VMIN > 0, VTIME = 0: This will make `read()` always wait for bytes (exactly how many is determined by `VMIN`), so `read()` could block indefinitely.

VMIN = 0, VTIME > 0: This is a blocking read of any number chars with a maximum timeout (given by `VTIME`). `read()` will block until either any amount of data is available, or the timeout occurs. This happens to be my favourite mode (and the one I use the most).

VMIN > 0, VTIME > 0: Block until either `VMIN` characters have been received, or `VTIME` after first character has elapsed. Note that the timeout for `VTIME` does not begin until the first character is received.

```
tty.c_cc[VTIME] = 10;    // Wait for up to 1s (10 deciseconds)
tty.c_cc[VMIN] = 0;
```

★ Baud Rate

```
// Set in/out baud rate to be 9600
cfsetispeed(&tty, B9600);
cfsetospeed(&tty, B9600);
```

```
B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400
```

```
B9600, B19200, B38400, B57600, B115200, B230400, B460800
```

⇒ If you are compiling with GNU C library, you can forget these enumeration and just specify an integer baud rate directly.

```
// Specifying a custom baud rate when using GNU C
cfsetispeed(&tty, 104560);
cfsetospeed(&tty, 104560);
```

★ Saving termios

```
// Save tty settings, also checking for error
if (tcsetattr(serial_port, TCSANOW, &tty) != 0) {
    printf("Error %i from tcsetattr: %s\n", errno, strerror(errno));
}
```

TCSANOW
the change occurs immediately.

TCSADRAIN
the change occurs after all output written to *fd* has been transmitted. This function should be used when changing parameters that affect output.

TCSAFLUSH
the change occurs after all output written to the object referred by *fd* has been transmitted, and all input that has been received but not read will be discarded before the change is made.

★ Reading and Writing

★ Writing

```
unsigned char msg[] = { 'H', 'e', 'l', 'l', 'o', '\r' };
write(serial_port, "Hello, world!", sizeof(msg));
```

★ Reading

```
// Allocate memory for read buffer, set size according to your needs
char read_buf [256];
memset(&read_buf, '\0', sizeof(read_buf));

// Read bytes. The behaviour of read() (e.g. does it block?,
// how long does it block for?) depends on the configuration
// settings above, specifically VMIN and VTIME
int n = read(serial_port, &read_buf, sizeof(read_buf));
```

★ Closing

```
close(serial_port)
```